Basics

- Open Source computing engine for parallel data processing on computer **Clusters**
- Written in *SCALA* and runs on a *Java Virtual Machine*
- **Cluster** (Group) of computers pools the resources of many machines together to use all cumulative resources as if they were a single computer
- *Spark* manages and coordinates the execution of tasks on data across a **Cluster** of computers
- **Cluster** of machines is managed by a Cluster Manager like *YARN*, or *Mesos*
- A **Spark Application** is submitted to these Cluster Managers, which will grant resources to the application

Spark Application

- **Spark Applications** consists of a **Driver** process and a set of **Executor** processes
- **Driver**:
    - Creates **SparkContext** and **SparkSession**
    - Runs the `main()` function, sits on a **Node** in the **Cluster**, and is responsible for three things:
        - Maintaining information about the **Spark Application**
        - Responding to a user's program or input
        - Analyzing, distributing, and scheduling work across the **Executors**
    - The **Driver** process is the heart of the **Spark Application** and maintains all relevant information during the lifetime of the application
- **Executor** (on **Worker Node**):
    - Responsible for carrying out the work that the **Driver** assigns them
    - Each **Executor** is responsible for 2 things:
        - Executing code in parallel (one per **Executor Core**) assigned to it by the **Driver**
        - Reporting the state of computation on that **Executor** back to the **Driver**

SparkSession

- **Driver** process that manages and controls the **Spark Application**
- Needs to be created in the application code when started through a standalone application

Lazy Evaluation

- *Spark* waits until the very last moment to execute the graph (**DAG**) of computation instructions
- In *Spark*, a plan of **Transformations** (**DAG**) is built up that is to be applied to the data rather than modifying the data immediately
- By waiting, *Spark* compiles this plan from the raw *DataFrame* **Transformations** to a streamlined physical plan (**DAG**) that will run as efficiently as possible across the **Cluster**
- As result, *Spark* can optimize the entire data flow from end to end (predicate pushdown)

Transformations

- Operation creating a new *DataFrame* from an existing one
- Narrow **Transformations**:
    - Operation where each **Partition** is used by at most one **Partition** in the output
    - No data shuffling between **Nodes**, thus more efficient
    - Executed in a single **Stage**
- Wide **Transformations**:
    - Operation where data from one **Partition** is used by multiple **Partitions** in the output
    - Require data shuffling across **Nodes**, thus expensive
    - Create new **Stage** in the **DAG**

Actions

- **Transformations** = logical **Transformation** plan (**DAG**)
- **Action** = triggers the computation
- An **Action** instructs *Spark* to compute a result from a series of **Transformations** (e.g., `count()`)
- Every **Action** starts a new **Job**

Job

- Series of parallel **Transformations** and **Actions** performed on a *DataFrame*
- Consists of:
    - **Stages**, each **Job** is divided into multiple **Stages** (set of operations)
    - **Tasks**, each **Stage** consists of multiple **Tasks** running in parallel on **Executor Cores**. Operate on **Partitions** of the *DataFrame*

Partitions

- To allow every **Executor** to perform work in parallel, *Spark* breaks up the data into **Partitions**
- **Partition**: Collection of rows that sit on one physical machine in the **Cluster**
- Represents how the data is physically distributed across the **Cluster** of machines during execution
- One **Partition** = Parallelism of one
- Many **Partitions** but only one **Executor** = Parallelism of one

Term	**Meaning**
DAG	○ Logical representation of a *Spark* computation ○ Consists of a series of **Transformations** (narrow/wide) applied to a *DataFrame* ○ Ensures that computations are structured efficiently and executed in the right order
Job	○ Complete computation triggered by an **Action** (e.g., `show()`, `collect()`) ○ Consists of multiple **Stages**
Stage	○ Collection of parallel **Tasks** separated by shuffle boundaries ○ Narrow **Transformations** can be executed in the same **Stage** ○ Wide **Transformations** (e.g., `groupBy()`) require data shuffling and create new **Stages**
Task	○ Each **Stage** is divided into multiple **Tasks**, where each **Task** runs on one **Partition** of the data ○ Single execution unit that processes one **Partition**

Component	Description
SparkContext	○ Core entry point for *Spark* operations ○ Created automatically inside a **SparkSession**
SparkSession	○ Entry point for a **Spark Application** ○ Provides access to *DataFrames*, *Datasets*, and *SQL* functionality ○ Created using: `SparkSession.builder.appName().getOrCreate()`
Cluster	○ Group of interconnected machines (**Nodes**) working together to process large-scale data in parallel ○ Managed by Cluster Manager like *YARN*, *Kubernetes*, or *Mesos*
Node	○ Virtual or physical machine in the **Cluster**
Driver Node	○ Virtual or physical machine where the **Driver** runs ○ Central coordinator of a **Spark Application** ○ Creates **SparkContext**, schedules tasks, and collects results from **Executors** ○ Allocates resources to **Executors** ○ Converts *Spark* code into a **DAG** of **Tasks**
Driver Core	○ Controls how many CPU cores are allocated to the **Spark Driver** for managing the **Spark Application** and scheduling **Tasks**
Driver Memory	○ RAM allocated to the **Spark Driver** for storing metadata, job information, and small collected results
Worker Node	○ Virtual or physical machine in the **Cluster** that contains and runs multiple **Executors** to process data in parallel
Executor	○ Process running on a **Worker Node** ○ Responsible for executing **Tasks** and storing intermediate data in memory or disk
Executor Core	○ Number of CPU cores allocated to each **Executor** ○ Defines how many **Tasks** an **Executor** can run in parallel
Executor Memory	○ RAM allocated to each **Executor** for performing computations and storing cached data

Parameter	Increase if	Typical Values
Executor Memory	OOM errors, large joins, caching required	4G-48G
Executor Cores	**Tasks** are too slow, but enough CPU available	4-5
Driver Memory	Large broadcast variables, many `.collect()` calls	4G-16G
Driver Cores	Many parallel **Driver** requests	1-4
Executor Instances	**Cluster** has free capacity, too few parallel **Tasks**	10-100