(Big) Data Engineering In Depth

Moustafa Mahmoud
Data Solution Architect

Chapter: Apache Spark

Course Objectives

- Understand the fundamentals of Apache Spark.
- Build data processing applications using Apache Spark.
- Understand Apache Spark APIs (RDD, DataFrames, and Datasets).
- Optimize and tune Apache Spark applications.
- Build streaming applications using Apache Spark.
- Build scalable machine learning applications using Apache Spark MLlib.
- Deploy Apache Spark in production environments.

Section: Course References

Course References

- Learning Spark, 2nd Edition By Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee.
- Spark: The Definitive Guide By Bill Chambers, Matei Zaharia.
- Random blog posts.

Section: Course Prerequisites

Course Prerequisites

Before beginning this course, participants should have:

- Experience in programming using Python.
- Basic programming skills using shell.
- An understanding of MapReduce foundations and Hive. Garage Education YouTube Playlist

Section: Python vs Scala

Python vs Scala

- Python is widely used with numerous tools and libraries available.
- Python is easier to learn than Scala; however, Scala might be more intuitive for those who prefer functional programming.
- Finding Python developers is generally easier for companies than finding Scala developers.
- Initially, Scala offered better performance in Apache Spark, but over time this advantage reduced, and now there's no big difference in speed.
- PySpark and Scala share the same Spark concepts, allowing for interchangeable use of examples from both languages without affecting learning.

Attention!

To Be a Spark Expert You Have to Be Able to Read a Little Scala Anyway![a]

[a] Referenced from High Performance Spark, 2nd Edition, Ch.01

Spark's Codebase and Documentation

- The quality of Spark's documentation is inconsistent.[1]
- Spark's codebase is very readable.
- Understanding the Spark codebase benefits **advanced users**.

[1] Referenced from High Performance Spark, 2nd Edition, Ch.01

Understanding Spark Through Scala

- Scala helps you understand Spark deeply.[2]
- Spark is written in Scala.
- To work with Spark's source code effectively, it's essential to understand (read) Scala.

[2] Referenced from High Performance Spark, 2nd Edition, Ch.01

RDD and Scala's Influence

- Scala's influence is evident in Spark's Resilient Distributed Datasets (RDD).[3]
- RDD methods are similar to Scala's collection tools.
- Functions like map, filter, and reduce are similar in both.
- Knowing Scala makes it easier to understand how RDDs work.

[3] Referenced from High Performance Spark, 2nd Edition, Ch.01

Spark as a Functional Framework

- Spark uses functional programming principles.
- Concepts like immutability and lambda are key.
- Understanding functional programming helps in using Spark well.[4]

[4] Referenced from High Performance Spark, 2nd Edition, Ch.01

Section: Introduction to Apache Spark

Introduction to Apache Spark

- Apache Spark was initiated at UC Berkeley in 2009, leading to the publication of Spark: Cluster Computing with Working Sets in 2010 by Matei Zaharia et al.

- Spark was developed to improve processing efficiency over Hadoop MapReduce, which struggled with iterative tasks because it launched separate jobs and reloaded data for each one. This was particularly important for machine learning algorithms that need multiple data passes.

- Initially, Spark was designed for batch applications, but it quickly expanded to include streaming, SQL analytics, graph processing, and machine learning.

Introduction to Apache Spark

- By 2013, the project had more than 100 contributors, and now it has over 2,000 contributors with more than 39,000 commits. It has been donated to the Apache Software Foundation, guaranteeing its future as a vendor-independent project.

- Key milestones in its development are Spark 1.0 in 2014, Spark 2.0 in 2016, and Spark 3.0 in 2020, highlighting its evolution and broad acceptance.

Introduction to Apache Spark

- Apache Spark is a **unified engine** designed for large-scale distributed data processing, either on-premises in data centers or in the cloud.
- Spark provides **in-memory** storage for intermediate computations, making it much faster than Hadoop MapReduce.
- Spark offers rich, composable APIs, for example:
  - Spark SQL: SQL for interactive queries.
  - MLlib: for machine learning over big data and complex computations.
  - Structured Streaming: for stream processing with near real-time data.
  - GraphX: for graph processing.

Section: About Databricks

About Databricks

- Databricks, founded by the early AMPlab team, joined the community to further develop Spark.
- The organization was founded to deliver a comprehensive analytics platform, streamlining Spark's application in big data processing and analytics.
- Databricks has bridged the gap between academic research and enterprise applications through its managed cloud service and significant contributions to the Spark project.

Beyond Apache Spark

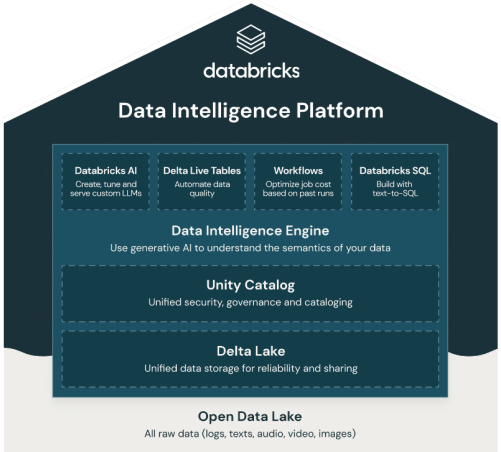


Photo copied from
https://www.datanami.com

Figure F-1: Databricks Analytics Platform

Do You Need Databricks to Work with Spark?

Do You Need Databricks to Work with Spark?

The answer is NO!

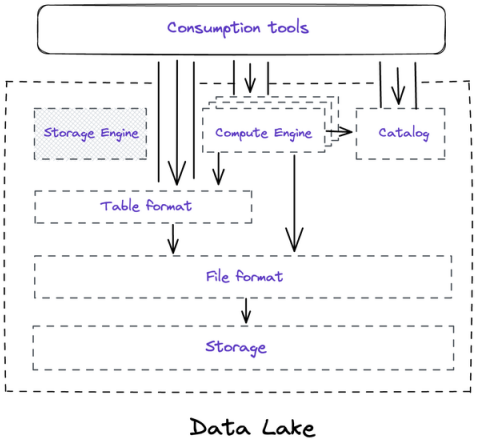
Databricks is one of the cloud options to run Apache Spark workload

Do You Need Databricks to Work with Spark?

- Spark workloads can be run both on the Cloud and on-premise.
  - **Cloud:** Choose any preferred cloud provider, like AWS, GCP, Databricks, or Azure.
  - **On-Premise:** Deploy on YARN, Mesos, or Kubernetes.
  - **Serverless Platforms:** For efficient resource management, consider options like:
    - **AWS:** EMR Serverless or AWS Glue.
    - **GCP:** Dataproc.
    - **Azure:** Azure Synapse.
    - **Databricks:** Databricks analytics platform.

Section: Apache Spark in Data Platforms

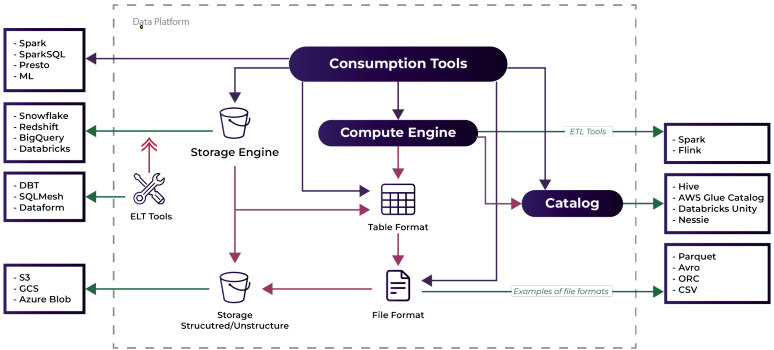
Technical components in a data lake



Apache Iceberg: The Definitive Guide:
Data Lakehouse Functionality,
Performance, and Scalability
on the Data Lake
PUBLISHED BY: O'Reilly Media, Inc.

Figure F-2: Technical components in a data lake

Technical components in a data lake



;

Section: Running Spark

Running Spark for Beginners

- **Databricks Community Edition:** The simplest option for Spark beginners. A free version that's easy to use for learning and small projects.
- **Install Spark Locally:** For hands-on experience with Spark's core features on your own machine.
- **Spark on Docker:** For a flexible, containerized environment that can replicate a production setup.

Note: For those new to Spark, starting with the Databricks Community Edition is highly recommended due to its user-friendly interface and comprehensive documentation.

Demo: Databricks Community Edition!

Demo: Install Spark Locally on Mac!

Demo: Install Spark Locally on Windows!

Demo: Install Spark Locally on Linux!

Demo: Spark Shell!

Demo: Spark SQL!

Section: From MapReduce to Apache Spark

The basic idea of MapReduce

- Assume we need to launch a high-throughput bulk-production sandwich shop.

[1]This example taken from https://reberhardt.com/cs110/summer-2018/lecture-notes/lecture-14/

The basic idea of MapReduce

- Assume we need to launch a high-throughput bulk-production sandwich shop.
- This sandwich has a lot of raw ingredients, and our target is to produce the sandwich as quickly as possible.

[1]This example taken from https://reberhardt.com/cs110/summer-2018/lecture-notes/lecture-14/

The basic idea of MapReduce

- Assume we need to launch a high-throughput bulk-production sandwich shop.
- This sandwich has a lot of raw ingredients, and our target is to produce the sandwich as quickly as possible.
- To make the production very quickly we need to distribute the tasks between the *workers*.

[1]This example taken from https://reberhardt.com/cs110/summer-2018/lecture-notes/lecture-14/

The basic idea of MapReduce

We break this into three stages

- Map.

[1]This example taken from https://reberhardt.com/cs110/summer-2018/lecture-notes/lecture-14/

The basic idea of MapReduce

We break this into three stages

- Map.
- Shuffle/Group (Mapper Intermediates).

[1]This example taken from https://reberhardt.com/cs110/summer-2018/lecture-notes/lecture-14/

The basic idea of MapReduce

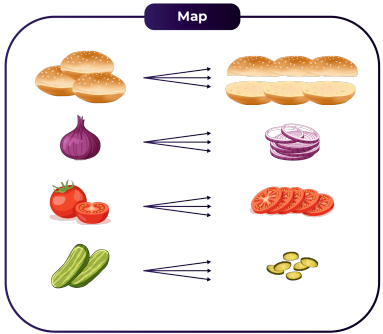
We break this into three stages

- Map.
- Shuffle/Group (Mapper Intermediates).
- Reduce

[1]This example taken from https://reberhardt.com/cs110/summer-2018/lecture-notes/lecture-14/

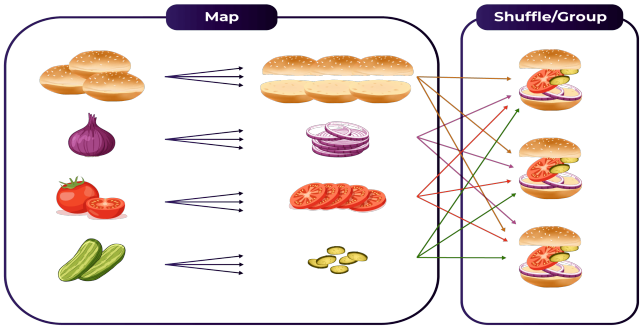
Map

We distribute our raw ingredients amongst the workers.



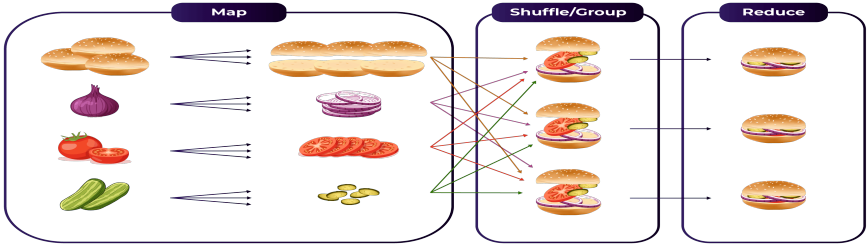
Shuffle/Group

We will organise and group the processed ingredients into piles, so that making a sandwich becomes easy.



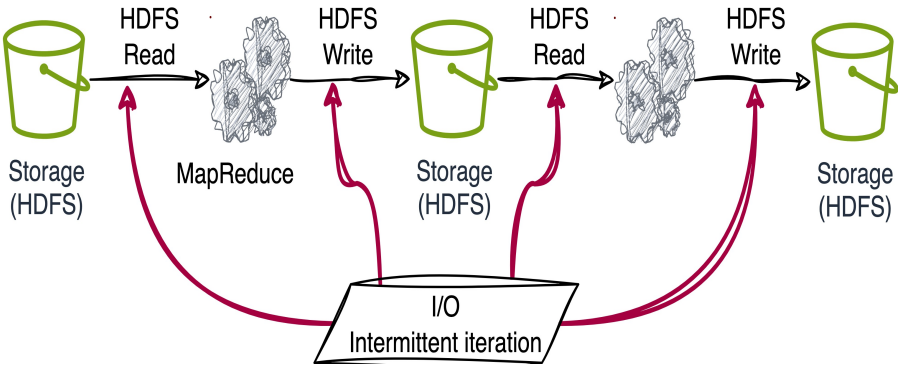
Reduce

we'll combine the ingredients into a sandwich



[1] This example taken from https://reberhardt.com/cs110/summer-2018/lecture-notes/lecture-14/

Map Reduce Bottelneck



Spark Motivation

- In-Memory Processing.
- Resilient Distributed Datasets (RDDs).
- Optimized Execution (DAG execution engine).
- Caching.
- Advanced Optimization (Catalyst optimizer and Tungsten).

Section: Spark Characteristics

Spark Characteristics

- Speed.
- Ease of use.
- Modularity.
- Extensibility.

Spark Characteristics: Speed

- Spark's speed is achieved through various strategies.
  - **Hardware Utilization:** Spark leverages commodity hardware with extensive memory and multiple cores, utilizing efficient multithreading and parallel processing for improved performance.
  - **Directed Acyclic Graph (DAG):** Spark constructs computations as a DAG. Its scheduler and optimizer create an efficient graph, allowing parallel task execution across cluster workers. *This topic will be discussed later.*

Spark Characteristics: Speed

- Spark's speed is achieved through various strategies.
  - **Tungsten Execution Engine:** Tungsten enhances Spark's speed by optimizing memory management, shifting from JVM-managed objects to direct binary processing, and employing cache-optimized algorithms and advanced code generation. These improvements reduce CPU and memory bottlenecks, significantly boosting performance. *This topic will be discussed later.*

Spark Characteristics: Ease of use

- Spark simplifies big data processing with an abstraction called a Resilient Distributed Dataset (RDD).
- RDDs serve as the foundation for higher-level data structures like DataFrames and Datasets in Spark. *This topic will be discussed later.*
- Compared with MapReduce and other complex distributed processing frameworks, Spark provides a simple programming model with a range of transformations and actions.

Spark Characteristics: Modularity

- Spark operations are flexible, supporting various workloads and languages: Scala, Java, Python, SQL, and R.
- It provides unified libraries with comprehensive APIs (SparkSQL, Structured Streaming, MLlib, and GraphX).
- These modules allow Spark to handle different workloads under a single engine.
- With Spark, you can develop a single application for diverse tasks without the need for separate engines or learning different APIs, achieving a unified processing framework.

Spark Characteristics: Extensibility

- Spark focuses on its fast, parallel computation engine rather than on storage (Unlike Apache Hadoop, which included both storage and compute).

Spark Characteristics: Extensibility

- Spark can process data from various sources like Hadoop, Cassandra, HBase, MongoDB, Hive, RDBMSs, and others in memory for faster processing.
- Spark's `DataFrameReaders` and `DataFrameWriters` allow Spark to interact with even more data sources, such as Kafka, Kinesis, Azure Storage, and Amazon S3.
- The Spark community maintains a list of third-party packages, enhancing its ecosystem with connectors for external data sources, performance monitors, and more.

Chapter: Apache Spark Distributed Execution

Spark Distributed Execution

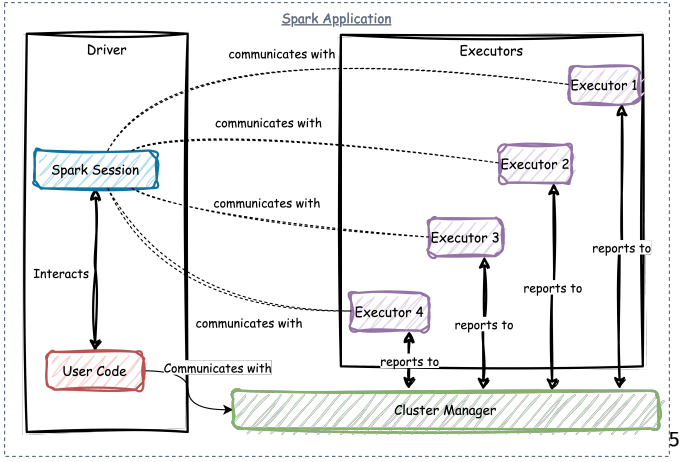
- Spark Application.
- Spark driver.
- Spark session.
- Cluster manager.
- Spark executors.
- Deployment mode.
- Data partition.

Section: Spark Application

Spark Distributed Execution: Spark Application

- A Spark application is a program designed for distributed data processing using Spark.
- It includes a driver program to run the main function and execute parallel operations on a cluster.
- The application divides processing tasks into smaller units called tasks, distributed across cluster nodes.
- Spark applications support various languages like Scala, Java, Python, and R.

Spark Application



[5]

[5]Spark the definitive guide, Ch.02

Section: Spark Driver

Spark Driver

The driver is the process `in the driver seat.`[6] of your Spark Application.

[6] Spark: The Definitive Guide, Chapter 15.

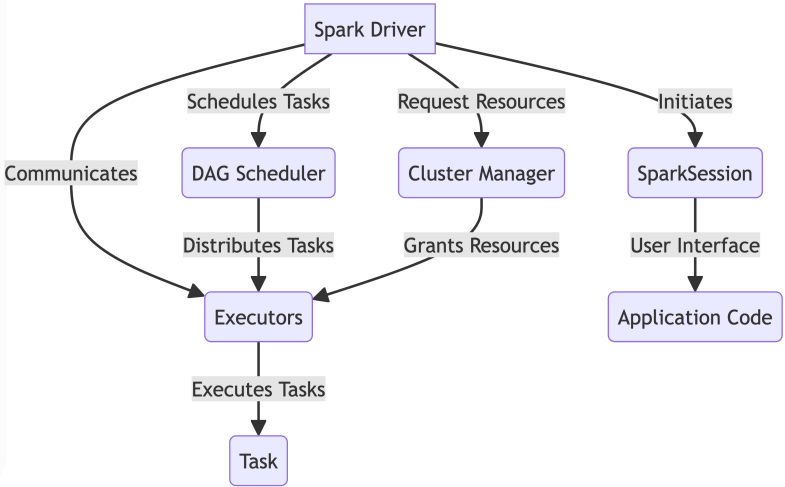
Spark Driver: Key functions

- It transforms all the Spark operations into DAG computations, schedules them, and distributes their execution as tasks across the Spark executors.
- Controlling the execution of a Spark Application.

Spark Driver: Key functions

- Acting as a process on a physical machine, responsible for the overall state of the application on the cluster.
- It instantiates the `SparkSession`
- It requests resources (CPU, memory, etc.) from the cluster manager for Spark's executors (JVMs).
- Once the resources are allocated, it communicates directly with the executors.

Spark Driver: Recap



Section: Spark Distributed Execution: SparkSession

What is a Session?

- A session refers to an interaction between two or more entities.
- In computing, it's especially common in networked computers on the internet.

Types of Sessions in Computing

- TCP session: A basic form of interaction in network communication.
- Login session: The period when a user is logged into a system.
- HTTP session: A series of interactions between a web server and a client.
- User session: The time a user interacts with a software application.

Introducing SparkSession

- Similar to the sessions mentioned, Spark has its own SparkSession.
- SparkSession provides a unified entry point to Spark's functionalities.

Functionality of SparkSession

- **SparkSession:** An object that provides a point of entry to interact with underlying Spark functionality.
- It allows programming Spark with its APIs.
- In an interactive Spark shell, the Spark driver instantiates a SparkSession for you.
- In a Spark application, you create a SparkSession object yourself.
- You can program Spark using DataFrame and Dataset APIs through SparkSession.
- In Scala and Python, the variable is available as `spark` when you start the console.

SparkSession

- The SparkSession instance is the way Spark executes user-defined manipulations across the cluster.
- There is a one-to-one correspondence between a SparkSession and a Spark Application.
- It connects the Spark driver program with the cluster manager.
- SparkSession determines the resource manager (YARN, Mesos, or Standalone) for communication.
- It allows configuration of Spark parameters.

Interacting with Spark in Earlier Versions

- In earlier versions of Spark, setting up a Spark application required creating a SparkConf and SparkContext.

```
1 from pyspark import SparkConf, SparkContext
2 from pyspark.sql import SQLContext
3
4 sparkConf = SparkConf().setAppName("SparkSessionExample").setMaster("local")
5 sc = SparkContext(conf=sparkConf)
6 sqlContext = SQLContext(sc)
7
```

Code C: Create SparkContext in old Spark versions

Interacting with Spark in Earlier Versions

```
1  //set up the spark configuration and create contexts
2  val sparkConf = new
       SparkConf().setAppName("SparkSessionZipsExample").setMaster("local")
3  val sc = new SparkContext(sparkConf)
4  sc.set("spark.some.config.option", "some-value")
5  val sqlContext = new org.apache.spark.sql.SQLContext(sc)
6
```

Code C: Scala: Create SparkContext in old Spark versions

Simplification in Spark 2.0 with SparkSession

- Spark 2.0 introduced SparkSession, simplifying the way you interact with Spark.
- SparkSession encapsulates SparkConf, SparkContext, and SQLContext.

```
1 from pyspark.sql import SparkSession
2
3 spark = SparkSession.builder \
4     .appName("SparkSessionExample") \
5     .config("spark.some.config.option", "value") \
6     .getOrCreate()
7
```

Code C: Pyspark: Create SparkSession

Simplification in Spark 2.0 with SparkSession

```
1  // Create a SparkSession. No need to create SparkContext.
2  val warehouseLocation = "file:${system:user.dir}/spark-warehouse"
3  val spark = SparkSession
4    .builder()
5    .appName("SparkSessionZipsExample")
6    .config("spark.sql.warehouse.dir", warehouseLocation)
7    .enableHiveSupport()
8    .getOrCreate()
9
```

Code C: Spark: Create SparkSession

Using SparkSession

- Spark 2.0 introduces SparkSession.
- With SparkSession, you can access all Spark functionalities.
- A unified entry point to Spark's functionality, reduces the need for multiple context initializations.
- Encapsulates the functionalities of SQLContext, HiveContext, and more.

Reference

- The examples shown are based on a blog post from Databricks.
- URL: https://www.databricks.com/blog/2016/08/15/how-to-use-sparksession-in-apache-spark-2-0.html

Section: Cluster manager

Introduction to Spark Cluster Managers

- The cluster manager allocates resources for Spark applications.
- Supports several managers: Standalone, Hadoop YARN, Apache Mesos, and Kubernetes.

Role of the Cluster Manager

- The Spark Driver and Executors do not exist in a void, and this is where the cluster manager comes in.
- The cluster manager is important for managing a cluster of machines intended to run Spark Applications.
- Maintains a `driver (or master)` and `worker` nodes, tied to **physical machines**.

Cluster Manager Components

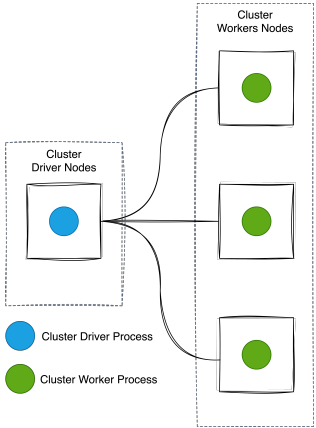


Figure F-3: A cluster driver and worker (no Spark Application yet).

Execution of Spark Applications

- The user requests resources from the cluster manager to initiate Spark applications.
- The user configures the application to specify resources for the driver or only for executors.
- The cluster manager directly manages the machines during the execution of the application.

Section: Execution Modes

Execution Modes Overview

- Execution modes define the location of resources when running Spark applications.
- Three modes available:
    1. Cluster mode
    2. Client mode
    3. Local mode

Cluster Manager Components

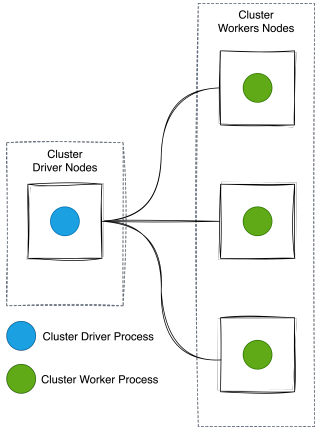


Figure F-4: A cluster driver and worker (no Spark Application yet).

Cluster Mode

- Most common mode for running Spark Applications.
- User submits a pre-compiled JAR, Python script, or R script to a cluster manager.
- The cluster manager then launches the driver process on a worker node inside the cluster.
- Executor processes also launched within the cluster.
- Cluster manager handles all Spark Application processes.
- This means that the cluster manager is responsible for maintaining all Spark Application–related processes.

Spark Cluster Mode

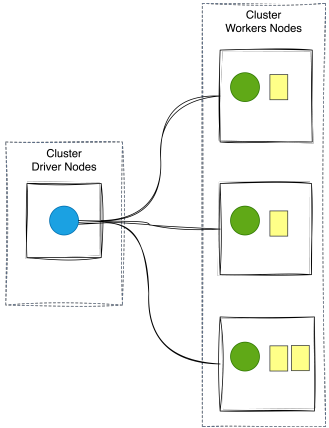


Figure F-5: Spark's cluster mode.

Client Mode

- Similar to cluster mode, but the **`Spark driver remains on the client machine that submitted the application`**.
- **Client machine** is responsible for maintaining the Spark driver process.
- **Cluster manager** maintains executor processes.
- Commonly used with gateway machines or edge nodes.
- The driver is running on a machine outside of the cluster but that the workers are located on machines in the cluster.

Spark Client Mode

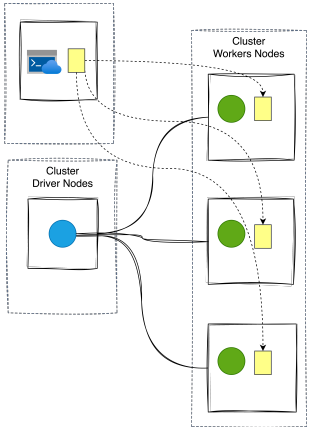


Figure F-6: Spark's client mode.

Local Mode

- Runs the entire application on a single machine.
- Parallelism achieved through threads on the same machine.
- Ideal for learning, testing, or local development.
- Not recommended for production use.

Section: Spark executors

Spark Executors in Depth

- Executors are processes that run the tasks assigned by the driver.
- Each Spark Application has distinct executor processes.
- Typically, one executor runs per node in most deployment modes.
- Executors' main function: Execute tasks, return status, and communicate outcomes.

Section: Data partition

Introduction to Data Distribution and Partitions

- **Data Distribution:** Physical data is distributed across storage as partitions residing in either HDFS or cloud storage.
- **Data Abstraction:** Spark treats each partition as a high-level logical data abstraction—as a DataFrame in memory.

Data Locality and Task Allocation

- **Data Locality:** Each Spark executor is preferably allocated a task that requires it to read the partition closest to it in the network, observing data locality.
- **Optimal Task Allocation:** Partitioning allows for efficient parallelism.
- **Minimize Network Bandwidth:** A distributed scheme of breaking up data into chunks or partitions allows Spark executors to process only data that is close to them, minimizing network bandwidth.

Benefits of Partitioning

- **Efficient Parallelism:** Partitioning allows executors to process data close to them.
- **Dedicated Processing:** Each core on an executor works on its own partition, minimizing network bandwidth usage.

Practical Example - Distributing Data

```
log_df = spark.read.text("path_to_large_text_file").repartition(8)
print(log_df.rdd.getNumPartitions())

```

Code C: Example SQL Query

This example splits data across clusters into eight partitions.

Practical Example - Creating a DataFrame

```
1  df = spark.range(0, 10000, 1, 8)
2  print(df.rdd.getNumPartitions())
3
```

Code C: Example SQL Query

This creates a DataFrame of 10,000 integers over eight partitions in memory.

Conclusion

- **Key Takeaway:** Efficient data partitioning is crucial for optimizing processing in Spark.

Section: Spark Operations

Spark Operations

- Spark **operations** on distributed data can be classified into two types: transformations and actions.
- All spark operations are `immutable`.

Immutable Objects

- An object whose state cannot change after it has been constructed is called immutable (unchangable).[7]
- The methods of an immutable object do not modify the state of the object.

[7] Referenced from https://otfried.org/courses/cs109scala/tutorial_mutable.html

Immutable Objects

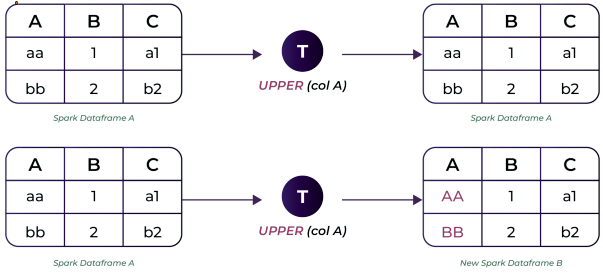


Figure F-7: Spark Dataframe is immutable, and you can't change its values.

Immutable Objects

Title

```python
from pyspark.sql import SparkSession

# Initialize a Spark session
spark = SparkSession.builder.appName("SimpleDataFrame").getOrCreate()

# Create a DataFrame
data = [("John", 28), ("Smith", 44), ("Adam", 65), ("Henry", 23)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns)

# Show the original DataFrame
df.show()

# filter rows where the age is greater than 30
filtered_df = df.filter(df.Age > 30)

# Show the transformed DataFrame
filtered_df.show()
```

Figure F-8: Filtering a PySpark DataFrame Based on Age

DEMO

DEMO

Spark Operations: Transformations

- Transformations: transform a Spark DataFrame into a new DataFrame *without altering the original data*.
- Example of Spark transformations, `map()`, `select()`, `filter()`, or `drop()`

Spark Transformations: What are Lazy Transformations?

- In Spark, transformations are *lazy*.
- This means computations are not executed immediately.
- Spark builds a **DAG** (Directed Acyclic Graph) of transformations.
- All Transformations results are not computed immediately, but they are recorded or remembered as a **lineage**.

Spark Transformations: Benefits of Lazy Evaluation

- **Optimization:** A lineage allows Spark, at a later time in its execution plan, to rearrange certain transformations, coalesce them, or optimize transformations into stages for more efficient execution.
- **Resource Management:** Executes tasks efficiently, using fewer resources.
- **Fault Tolerance:** Easier to recompute parts of the pipeline if a part fails.

Spark Transformations: Lazy Transformation

- Consider a dataset with map and filter transformations.
- Spark does not execute these transformations when they are defined.
- Transformations are executed when an action (like **collect**, **count**) is called.

Lazy Transformations Example

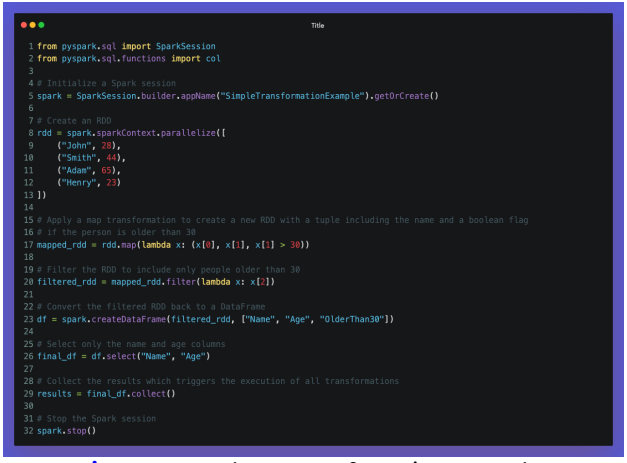


Figure F-9: Spark Lazy Transformations Example.

Spark Operations: Actions

- An action triggers the lazy evaluation of all the recorded transformations.
- Actions are operations that trigger execution of transformations.
- They are used to either compute a result to be returned to the Spark driver program or to write data to an external storage system.
- Actions include operations like **count**, **collect**, **saveAsTextFile**, and **take**.

Examples of Spark Actions

- **collect()**: Collects all elements from the Spark context to the driver program.
- **count()**: Returns the number of elements in the dataset.
- **saveAsTextFile(path)**: Saves the dataset to a text file at the specified path.
- **take(n)**: Returns an array with the first n elements of the dataset.

Section: Narrow and Wide Transformations

Introduction to Spark Transformations

- Transformations create new RDDs from existing ones.
- Spark has two types of transformations: Narrow and Wide.

What are Narrow Transformations?

- Transformations that do not require data shuffling between partitions.
- Examples: `map()`, `filter()`.
- Data processing is limited to a single partition.

What are Narrow Transformations?

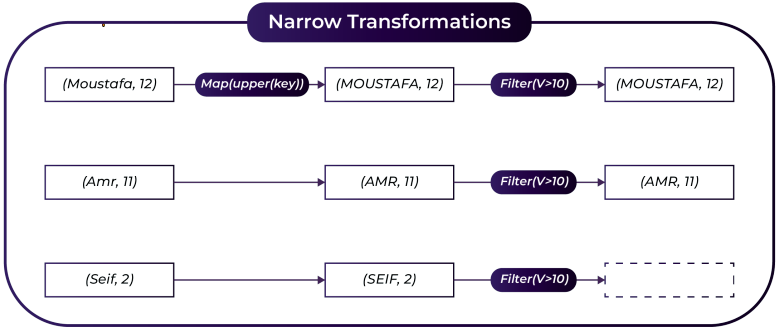


Figure F-10: Spark Narrow Transformations.

Benefits of Narrow Transformations

- Efficient with minimal data movement.
- Best for independent data processing tasks.

What are Wide Transformations?

- Transformations that involve shuffling data across partitions.
- Examples: `groupBy()`, `reduceByKey()`.

What are Wide Transformations?

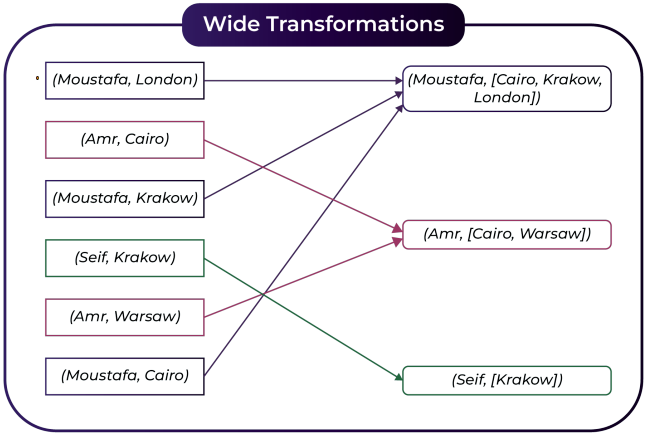


Figure F-11: Spark Wide Transformations.

Wide Transformations and Dependencies

- **Wide Dependencies**: Require data from multiple partitions, often involving shuffling.
- **Examples**: `groupBy()`, `orderBy()` - data is combined across partitions, affecting performance.
- **Impact**: These transformations are necessary for operations like counting occurrences across a dataset.

Implications of Wide Transformations

- Shuffling can be expensive in terms of time and network I/O.
- Essential for aggregation and grouping operations.

Narrow vs. Wide Dependencies

- **Narrow Dependencies**: A single output partition can be computed from a single input partition without data exchange.
- **Examples**: `filter()`, `contains()` - operate independently on partitions.

Section: Repartition vs. Coalesce

Repartition vs. Coalesce

- In Apache Spark, repartition and coalesce are two methods used to change the number of partitions in an RDD (Resilient Distributed Dataset):

Repartition vs. Coalesce

Aspect	Repartition	Coalesce
Purpose	Increases or decreases the number of partitions.	Decreases the number of partitions.
Mechanism	Shuffles all the data across the network to create a new set of partitions.	Merges existing partitions without a full data shuffle.
Use Case	Ideal for increasing the number of partitions or significantly changing the distribution of data.	Efficient for reducing the number of partitions when the target number is less than the current number.
Cost	Expensive due to the full data shuffle.	Less expensive than `repartition` as it minimizes data movement.

Table T-1: Comparison of Repartition and Coalesce in Apache Spark

High-Level Code: Repartition

```
1  def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T] = withScope {
2   coalesce(numPartitions, shuffle = true)
3  }
4
```

Code C: Example SQL Query

- **Key Points**:
  - **Shuffle**: Always performs a shuffle by calling ‘coalesce‘ with ‘shuffle = true‘.
  - **Usage**: Suitable for both increasing and decreasing the number of partitions with even data distribution.

High-Level Code: Repartition

- **Operation**: Calls ‘coalesce‘ with ‘shuffle = true‘.
- **Process**: Full data shuffle across the network to redistribute data evenly.
- **Impact on Logs**:
  - Increased log entries due to extensive shuffle operations.
  - Higher overhead and cost.
  - Useful for balancing load and increasing parallelism.

High-Level Code: Repartition

- Shuffling Logic (If shuffle is true):
  - Data is shuffled to distribute it evenly across the new partitions.
  - Each item is assigned a new partition based on a random function.
  - A new RDD is created with the shuffled data and the specified number of partitions.

High-Level Code: Coalesce

```
def coalesce(numPartitions: Int, shuffle: Boolean = false,
             partitionCoalescer: Option[PartitionCoalescer] = Option.empty)
            (implicit ord: Ordering[T] = null): RDD[T] = withScope {
 require(numPartitions > 0, s"Number of partitions ($numPartitions) must be positive.")
  if (shuffle) {
   // Shuffle logic
   val distributePartition = (index: Int, items: Iterator[T]) => {
    var position = new XORShiftRandom(index).nextInt(numPartitions)
    items.map { t =>
     position = position + 1
     (position, t)
    }
   } : Iterator[(Int, T)]
   new CoalescedRDD(new ShuffledRDD[Int, T, T](
     mapPartitionsWithIndexInternal(distributePartition, isOrderSensitive = true),
     new HashPartitioner(numPartitions)),
    numPartitions,
    partitionCoalescer).values
  } else {
   // No-shuffle logic
   new CoalescedRDD(this, numPartitions, partitionCoalescer)
  }
}
```


High-Level Code: Coalesce

- **Key Points**:
  - **Shuffle**: Optional; can perform a shuffle if ‘shuffle = true‘.
  - **No Shuffle**: Default behavior (without shuffle) merges partitions locally.
  - **Usage**: Efficient for reducing the number of partitions without shuffling.

High-Level Code: Coalesce

- **Operation**: Can operate with or without shuffle.
- **Process**:
  - **No Shuffle (default)**: Merges partitions locally.
  - **With Shuffle**: Performs a shuffle for even data distribution if specified.
- **Impact on Logs**:
  - Decreased log entries when merging locally without shuffle.
  - Lower overhead and cost.
  - Efficient for reducing partitions, especially useful for write operations.
  - Optional shuffle provides flexibility for balanced distribution.

Section: Understanding Spark Application Concepts

Understanding Spark Application Concepts

- **Job**: A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g., save(), collect()).
- **Stage**: Each job gets divided into smaller sets of tasks called stages that depend on each other.
- **Task**: A single unit of work or execution that will be sent to a Spark executor.

Spark Jobs

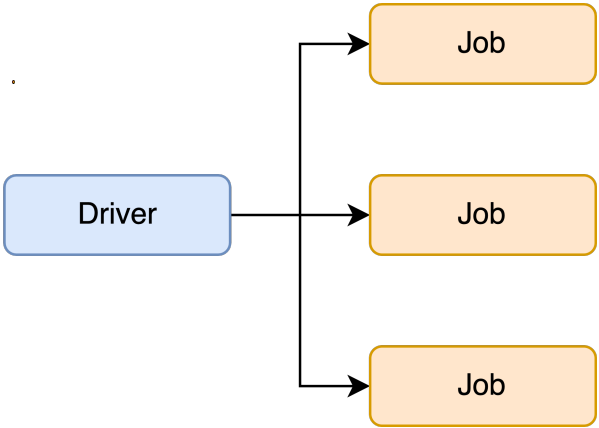


Figure F-12: Spark driver creating one or more Spark jobs.

Spark Stages

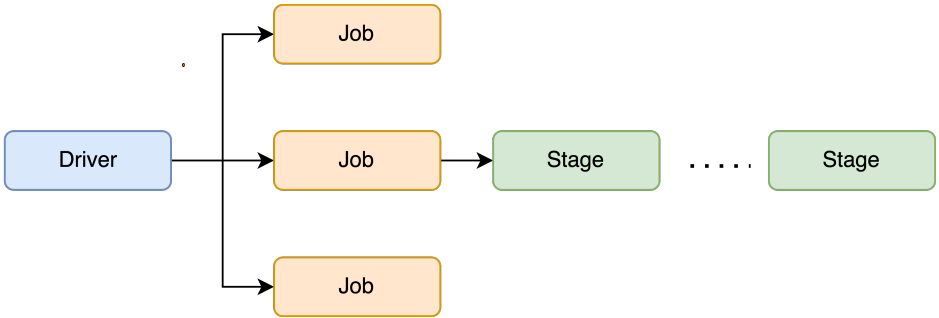


Figure F-13: Spark job creating one or more stages

Spark Stages

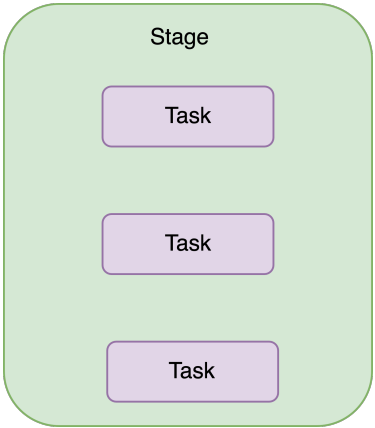


Figure F-14: Spark stage creating one or more tasks to be distributed to executors

Spark Tasks

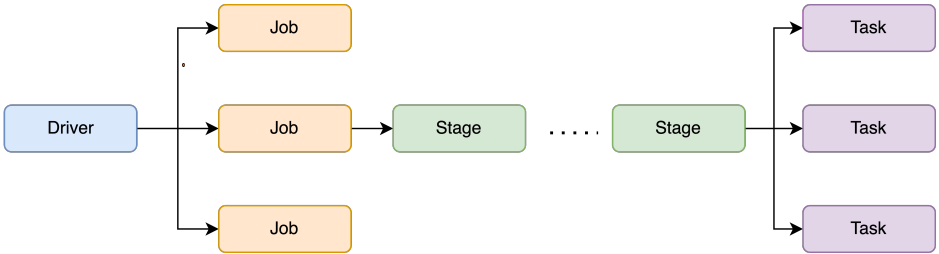


Figure F-15: Spark stage creating one or more tasks to be distributed to executors

Section: Running Word Count & Aggregation using Spark

Section: Spark Application Life Cycle

The Life Cycle of a Spark Application (Inside Spark)

- The Life Cycle of a Spark Application (Inside Spark)

The Life Cycle of a Spark Application (Outside Spark)

- The Life Cycle of a Spark Application (Outside Spark)

The Life Cycle of a Spark Application (Inside Spark)

- The Life Cycle of a Spark Application (Inside Spark)

ADVANCED: SPARK DRIVER INTERNAL SCHEDULER

- GOING DEEPER INTO SPARK DRIVER.
- CAN WE OPTIMIZE SPARK DRIVER WORKLOAD?

Section: Further Readings and Assignment

Thank you for watching!

See you in the next video ☺