SEAS 8515 - Lecture 2
Introduction to Apache Spark

School of Engineering
& Applied Science
THE GEORGE WASHINGTON UNIVERSITY

Dr. Adewale Akinfaderin

Genesis of Spark

- Genesis of Apache Spark: Apache Spark emerged as a unified processing engine for big data, driven by the need for efficient handling of large-scale data processing.
- Influence of Google's Scale: Google's search engine, known for its immense scale in indexing and searching the world's internet data rapidly, served as a significant inspiration for developments in big data processing.
- Google's Unique Naming: Google, a misspelling of "googol" (1 followed by 100 zeros), symbolizes its commitment to managing massive scales of data.
- Limitations of Traditional Systems: Conventional storage systems, like RDBMS, and traditional programming methods were inadequate for the vast scale at which Google intended to index and search internet documents.
- Innovation at Google: This challenge led to the creation of three key technologies: the Google File System (GFS), MapReduce (MR), and Bigtable.

Genesis of Spark

- Google File System (GFS): GFS was designed as a fault-tolerant, distributed filesystem to support large clusters of commodity hardware.
- Bigtable: This system provided scalable storage for structured data across the GFS infrastructure.
- MapReduce: Introducing a parallel programming model based on functional programming, MR was crucial for processing large-scale data distributed over GFS and Bigtable.
- MapReduce Operations: MR applications run computations (map and reduce functions) directly where data is stored, minimizing network traffic and emphasizing data locality and rack affinity in cluster setups.
- Influence on Open Source Community: While Google's work was proprietary, their concepts influenced the open-source domain, particularly Yahoo!, which faced similar big data challenges for its search engine, leading to further innovations like Apache Spark.

Genesis of Spark

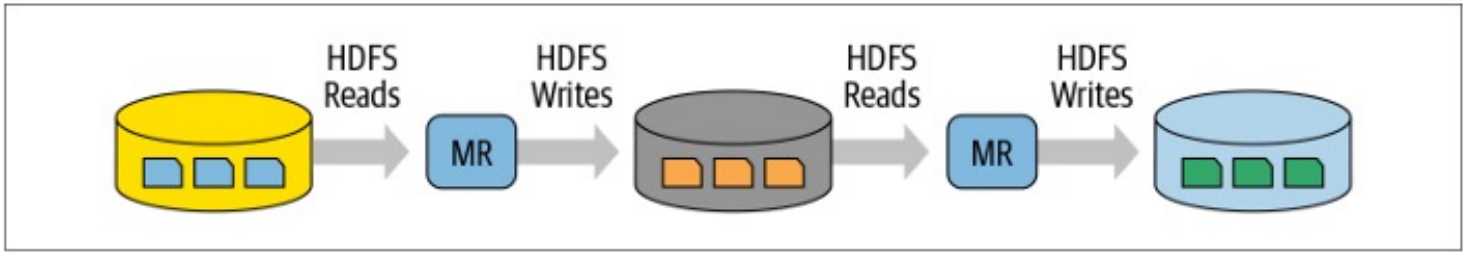
- Blueprint from Google's GFS: The Hadoop File System (HDFS) and MapReduce framework were inspired by the computational solutions in Google's GFS paper.
- Apache Hadoop Development: In April 2006, Hadoop was donated to the Apache Software Foundation, leading to a framework comprising Hadoop Common, MapReduce, HDFS, and Apache Hadoop YARN.
- Widespread Adoption: Beyond Yahoo!, Apache Hadoop gained significant popularity, influencing a large open-source community and leading to the creation of companies like Cloudera and Hortonworks (now merged).
- MapReduce Shortcomings: Despite its success, Hadoop's MapReduce framework was challenging to manage and had operational complexities.
- Verbose API and Fault Tolerance Issues: The general batch-processing MapReduce API was seen as verbose, requiring extensive setup code and having brittle fault tolerance.

Genesis of Spark

- Performance Issues with Disk I/O: Large-scale data jobs in Hadoop involved frequent disk I/O for intermediate results, leading to long-running tasks, sometimes extending to days.
- Limitations for Varied Workloads: Hadoop MapReduce was not well-suited for diverse workloads like machine learning, streaming, or interactive SQL-like queries.
- Development of Bespoke Systems: To address these limitations, specialized systems like Apache Hive, Storm, Impala, Giraph, Drill, and Mahout were developed, each with unique APIs and cluster configurations.
- Increased Complexity and Learning Curve: These new systems added to Hadoop's operational complexity and steepened the learning curve for developers.

Genesis of Spark

Intermittent iteration of reads and writes between map and reduce computations



Spark – Early Years

1. **Spark's Development**: Developed at UC Berkeley's RAD Lab in 2009, Spark aimed to improve upon Hadoop MapReduce's inefficiencies and complexity.
2. **Performance Advantages**: Early tests showed Spark was 10 to 20 times faster than Hadoop MapReduce, with even greater gains over time.
3. **Enhanced Features**: Spark enhanced Hadoop's ideas, offering high fault tolerance, parallel processing, in-memory storage, and user-friendly APIs.
4. **Widespread Adoption and Release**: Gaining popularity by 2013, Spark was donated to the ASF, and Apache Spark 1.0 was released in May 2014, signaling a new phase of development and enhancements.

What is Apache Spark?

- **Unified Engine**: Apache Spark is a unified processing engine for large-scale data, suitable for both on-premises data centers and cloud environments, focusing on distributed data processing.
- **In-Memory Storage**: Spark's in-memory storage for intermediate computations significantly speeds up processing, outperforming traditional systems like Hadoop MapReduce.
- **Diverse Libraries and APIs**: Spark encompasses various libraries with composable APIs for different tasks: MLlib (machine learning), Spark SQL (interactive queries), Structured Streaming (real-time data processing), and GraphX (graph processing).
- **Design Philosophy**: Spark is built around four key principles: speed (for fast processing), ease of use (user-friendly design), modularity (flexible and scalable architecture), and extensibility (capability to expand and integrate).

Apache Spark - Speed

Spark's speed is driven by its use of advanced hardware (like multi-core processors and large memory), an efficient directed acyclic graph (DAG) for query optimization and parallel execution, and its Tungsten execution engine, which minimizes disk I/O through whole-stage code generation and memory-efficient processing.

Apache Spark – Ease of Use

Spark achieves simplicity by providing a fundamental abstraction of a simple logical data structure called a Resilient Distributed Dataset (RDD) upon which all other higher-level structured data abstractions, such as DataFrames and Datasets, are constructed. By providing a set of transformations and actions as operations, Spark offers a simple programming model that you can use to build big data applications in familiar languages.

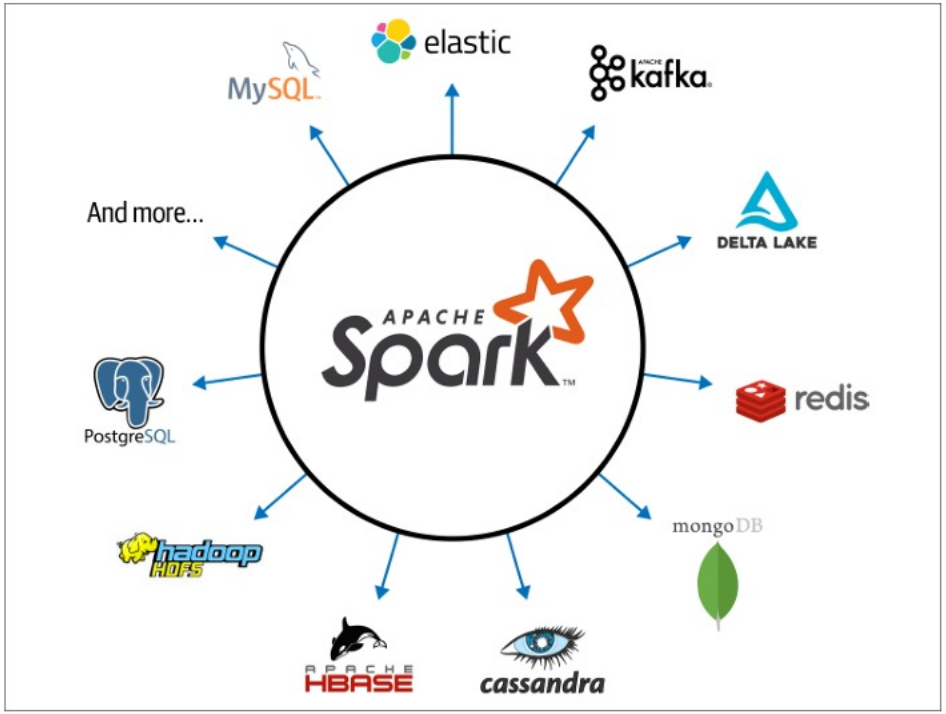
Apache Spark – Modularity

Spark supports a wide range of workloads and can be used with various programming languages, including Scala, Java, Python, SQL, and R. Its unified libraries provide well-documented APIs across key modules like Spark SQL, Spark Structured Streaming, Spark MLlib, and GraphX. This unified approach allows for writing a single Spark application that handles different workloads without needing separate engines or learning different APIs, streamlining the processing of diverse tasks under one engine.

Apache Spark – Extensibility

Spark, focusing on fast parallel computation, decouples storage and compute unlike Apache Hadoop, enabling it to process data from a variety of sources like Hadoop, Cassandra, and MongoDB in memory. Its capabilities are extendable to additional sources such as Kafka and Amazon S3, supported by a growing ecosystem of third-party packages for connectors and performance tools.

Apache Spark – Extensibility and Connector Ecosystem

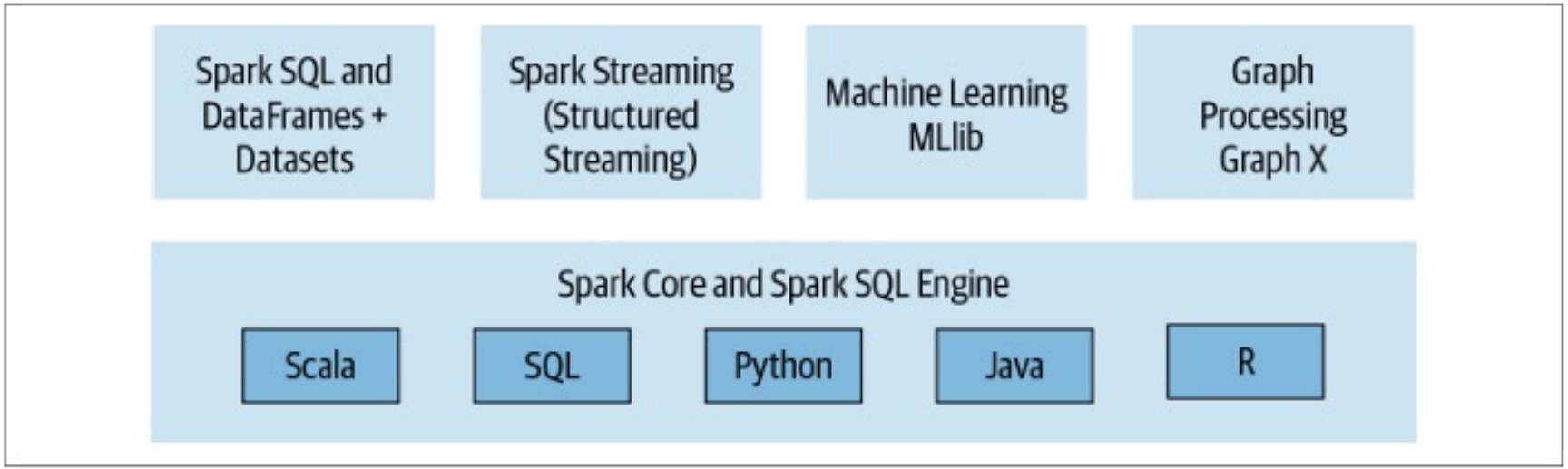


Apache Spark Components as a Unified Stack

Diverse Libraries for Workloads: Spark offers four key components as libraries to handle various types of data processing: Spark SQL for interactive queries, Spark MLlib for machine learning, Spark Structured Streaming for real-time data processing, and GraphX for graph processing.

Unified Execution Engine: These components operate on top of Spark's core fault-tolerant engine. Regardless of the language used (Java, R, Scala, SQL, Python), Spark applications are written using APIs. These APIs transform the code into a Directed Acyclic Graph (DAG), which is then executed as compact bytecode across the cluster's Java Virtual Machines (JVMs).

Apache Spark Components as a Unified Stack



Spark SQL

Structured Data Processing: Spark SQL effectively manages structured data, supporting a variety of formats (CSV, JSON, Avro, ORC, Parquet) and RDBMS tables. It allows the creation of both temporary and permanent tables in Spark, utilizing SQL-like queries in various languages.

Language Agnostic and Performance: Spark SQL is ANSI SQL:2003-compliant and acts as a standalone SQL engine. Code written in Scala, Python, R, or Java produces identical bytecode, ensuring consistent performance across these languages when executing SQL queries on Spark DataFrames.

Spark SQL

For example, in this Scala code snippet, you can read from a JSON file stored on Amazon S3, create a temporary table, and issue a SQL-like query on the results read into memory as a Spark DataFrame:

```
// In Scala
// Read data off Amazon S3 bucket into a Spark DataFrame
spark.read.json("s3://apache_spark/data/committers.json")
  .createOrReplaceTempView("committers")
// Issue a SQL query and return the result as a Spark DataFrame
val results = spark.sql("""SELECT name, org, module, release, num_commits
    FROM committers WHERE module = 'mllib' AND num_commits > 10
    ORDER BY num_commits DESC""")
```

Spark MLlib

MLlib as Spark's ML Library: Apache Spark's MLlib is a library containing a variety of common machine learning algorithms. Its performance has significantly improved with enhancements in Spark 2.x. MLlib is built on top of DataFrame-based APIs, facilitating the development of machine learning models.

MLlib Structure and Functionality: Since Apache Spark 1.6, MLlib is divided into two packages: spark.mllib (the RDD-based API, now in maintenance mode) and spark.ml (the DataFrame-based API, receiving new features). MLlib, as a comprehensive machine learning library in Spark, supports feature extraction, transformation, pipeline building, model persistence, linear algebra operations, statistics, and includes low-level ML primitives like gradient descent optimization.

Spark MLlib

```
# In Python
from pyspark.ml.classification import LogisticRegression
...
training = spark.read.csv("s3://...")
test = spark.read.csv("s3://...")

# Load training data
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(training)

# Predict
lrModel.transform(test)
...
```

Spark Structured Streaming

Spark 2.0's New Feature: Apache Spark 2.0 introduced Continuous Streaming and Structured Streaming APIs, based on the Spark SQL engine and DataFrame APIs.

Structured Streaming Maturity: By Spark 2.2, Structured Streaming became stable for production use, integrating real-time and static data processing.

Stream as a Growing Table: The model treats a data stream like a continuously expanding table, enabling straightforward querying like a regular table.

Ease of Streaming Application Development: Structured Streaming simplifies the development of streaming applications, handling fault tolerance and late-data semantics, and extends support to various data sources like Kafka and Kinesis.

Spark Structured Streaming

The following code snippet shows the typical anatomy of a Structured Streaming application. It reads from a localhost socket and writes the word count results to an Apache Kafka topic:

```
# In Python
# Read a stream from a local host
from pyspark.sql.functions import explode, split
lines = (spark
  .readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load())

# Perform transformation
# Split the lines into words
words = lines.select(explode(split(lines.value, " ")).alias("word"))

# Generate running word count
word_counts = words.groupBy("word").count()

# Write out to the stream to Kafka
query = (word_counts
  .writeStream
  .format("kafka")
  .option("topic", "output"))
```

GraphX

GraphX Library Functionality: GraphX is a specialized library in Apache Spark designed for manipulating various types of graphs (like social networks or network topologies) and executing graph-parallel computations.

Standard Algorithms and Community Contributions: It includes a suite of standard graph algorithms for tasks such as analysis, connection finding, and traversal. Key algorithms available in GraphX include PageRank, Connected Components, and Triangle Counting, contributed by the user community.

GraphX

This code snippet shows a simple example of how to join two graphs using the GraphX APIs:

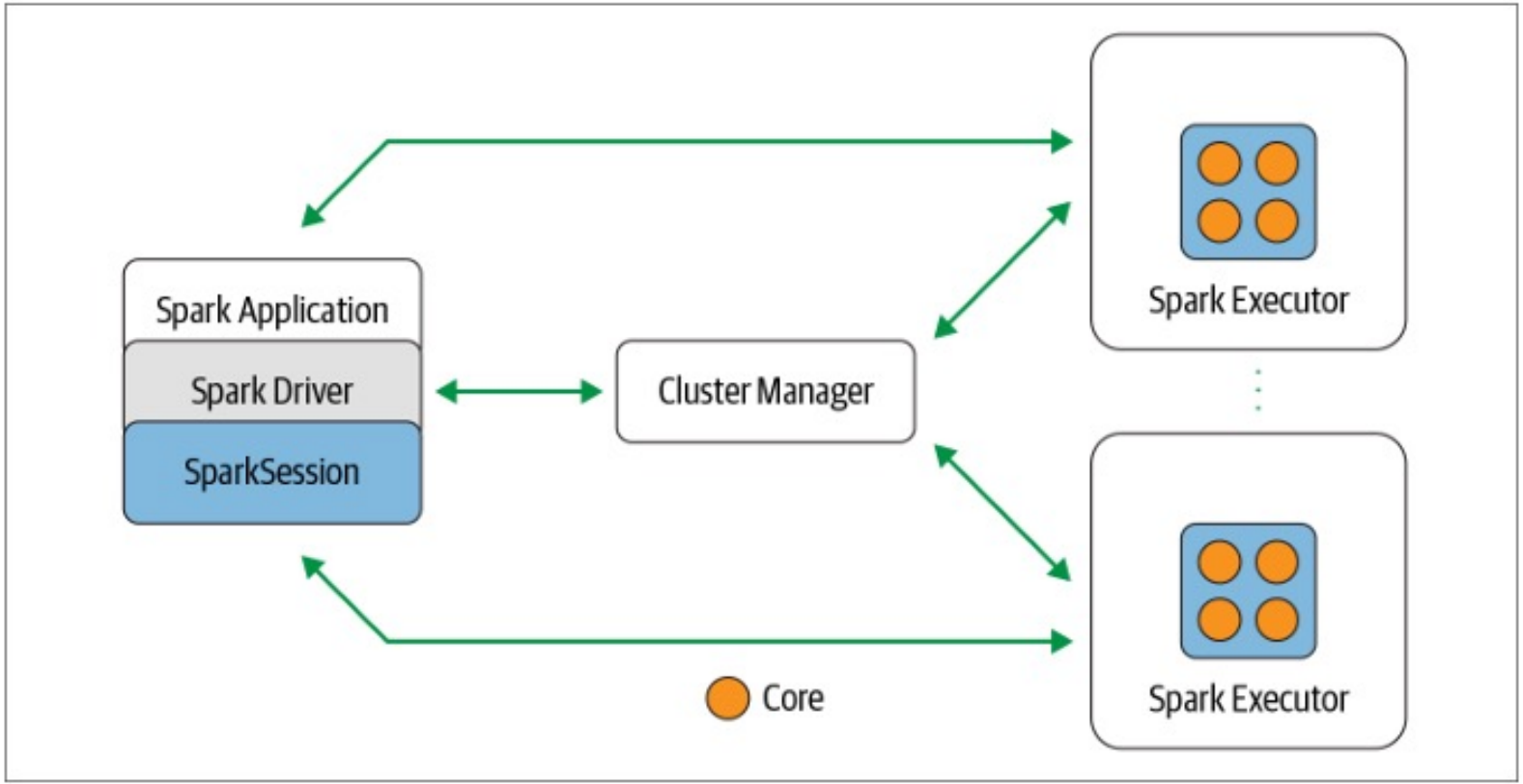
```
// In Scala
val graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
val graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

Apache Spark’s Distributed Execution

Distributed Nature of Spark: Apache Spark functions as a distributed data processing engine, operating across a cluster of machines, with its various components collaboratively executing tasks.

Spark's Architecture Components: In Spark's architecture, the driver program orchestrates parallel operations, while the SparkSession facilitates communication with the distributed components, like executors and the cluster manager, in the cluster.

Apache Spark components and architecture



Spark driver

Driver's Roles in Spark Application: In a Spark application, the driver is crucial for initializing a SparkSession. Its roles include communicating with the cluster manager, requesting resources (like CPU and memory) for Spark's executors (JVMs), and transforming Spark operations into Directed Acyclic Graph (DAG) computations.

Scheduling and Resource Allocation: The driver also schedules and distributes these computations as tasks across the Spark executors. Once resources are allocated, the driver directly communicates with the executors to manage task execution.

SparkSession

SparkSession as a Unified Interface: In Spark 2.0, SparkSession was introduced as a unified entry point, consolidating various older contexts like SparkContext and SQLContext, thereby simplifying user interaction with Spark's functionalities.

Enhanced Simplicity and Usability: By integrating multiple contexts into a single SparkSession, Spark 2.x significantly streamlined the user experience, making working with Spark more straightforward and reducing the complexity of the code.

SparkSession

Preservation of Backward Compatibility: Despite these integrations, Spark 2.x maintains backward compatibility, allowing legacy code using SparkContext or SQLContext from Spark 1.x to remain operational.

Reduction in Boilerplate Code: The SparkSession in Spark 2.x enables the creation of a single session per JVM, which can be utilized to execute a variety of operations, thus minimizing the need for separate contexts for different tasks like streaming or SQL, and reducing the boilerplate code in Spark applications.

SparkSession

```scala
// In Scala
import org.apache.spark.sql.SparkSession

// Build SparkSession
val spark = SparkSession
  .builder
  .appName("LearnSpark")
  .config("spark.sql.shuffle.partitions", 6)
  .getOrCreate()
...
// Use the session to read JSON
val people = spark.read.json("...")
...
// Use the session to issue a SQL query
val resultsDF = spark.sql("SELECT city, pop, state, zip FROM table_name")
```

SparkSession

```
#In Python

from pyspark.sql import SparkSession

# Build SparkSession
spark = SparkSession.builder \
    .appName("LearnSpark") \
    .config("spark.sql.shuffle.partitions", "6") \
    .getOrCreate()

# Use the session to read JSON
people = spark.read.json("path_to_json_file")

# Use the session to issue a SQL query
resultsDF = spark.sql("SELECT city, pop, state, zip FROM table_name")
```

Cluster Manager and Spark Executor

Cluster Manager Role: The cluster manager in Apache Spark is vital for managing and allocating resources across the cluster where the Spark application runs. Spark supports several types of cluster managers, including its built-in standalone cluster manager, Apache Hadoop YARN, Apache Mesos, and Kubernetes.

Spark Executor Functionality: In a Spark cluster, each worker node runs a Spark executor. These executors are in charge of executing tasks assigned to them and communicate with the driver program. Typically, in most deployment modes, each node in the cluster runs a single executor.

Deployment Modes

Versatile Deployment Options: Spark's ability to support a wide range of deployment modes is a key feature, allowing it to adapt to various configurations and environments. This flexibility is due to the cluster manager's agnosticism to its operational environment, as long as it can manage Spark's executors and meet resource demands.

Compatibility with Popular Environments: Spark's compatibility with popular environments like Apache Hadoop YARN and Kubernetes allows it to function effectively in different modes. This adaptability ensures that Spark can be deployed in a variety of settings, catering to different operational requirements and infrastructure setups.

Cheat sheet for Spark deployment modes

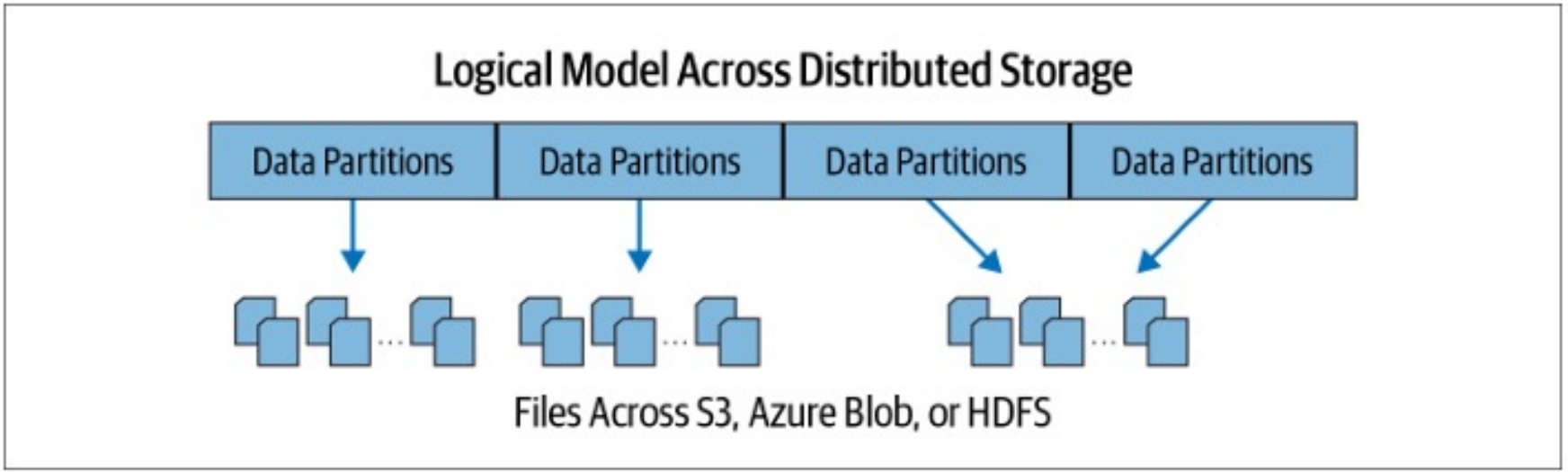
Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

Distributed data and partitions

Data Distribution and Partitioning: In Apache Spark, physical data is distributed and stored in partitions across systems like HDFS or cloud storage, with each partition treated as a logical DataFrame in memory.

Data Locality Optimization: Spark optimizes processing by aligning tasks with data locality, ideally assigning tasks to executors based on the proximity of the data partition within the network to minimize data transfer times.

Distributed data and partitions



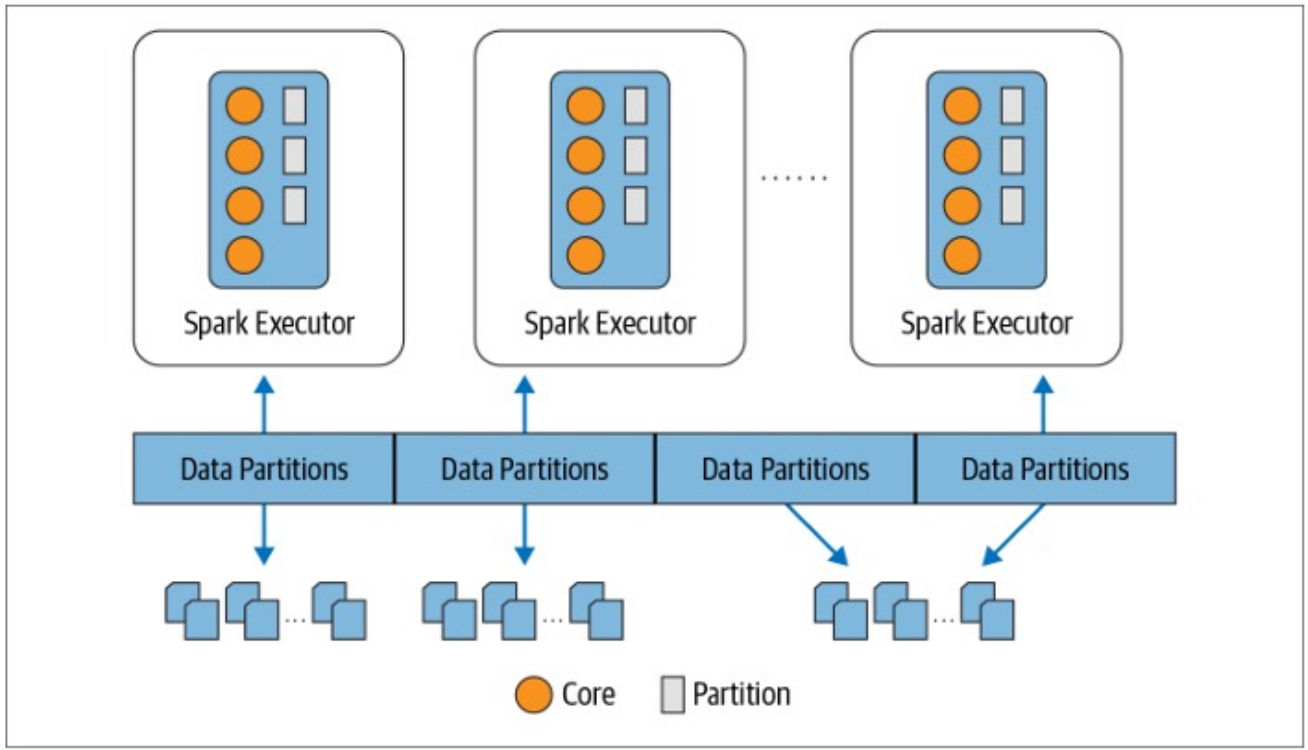
Distributed data and partitions

Efficient Parallel Processing via Partitioning: Spark's data partitioning enables efficient parallel processing, as it distributes data into chunks that allow executors to work on data local to them, thereby minimizing network bandwidth usage and optimizing performance.

Each executor’s core gets a partition of data to work on ...

Distributed data and partitions

Each executor's core gets a partition of data to work on



Distributed data and partitions

For example, this code snippet will break up the physical data stored across clusters into eight partitions, and each executor will get one or more partitions to read into its memory:

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

And this code will create a DataFrame of 10,000 integers distributed over eight partitions in memory:

```
# In Python
df = spark.range(0, 10000, 1, 8)
print(df.rdd.getNumPartitions())
```

Both code snippets will print out 8.

Who uses Spark? Data Science

Data Science in the Big Data Era: Data science involves cleansing, exploring, and modeling data to narrate stories, requiring skills in statistics, mathematics, and programming languages like R and Python.

Tool Proficiency for Data Scientists: They often use analytical tools such as SQL and libraries like NumPy and pandas, focusing on data transformation and familiar classification, regression, and clustering algorithms.

Supportive Features of Apache Spark: Spark caters to the iterative and experimental nature of data science tasks with its MLlib for machine learning algorithms, Spark SQL for data exploration, and enhanced support for deep learning models and GPU resources in recent versions.

Who uses Spark? Data Engineering

Collaboration Post-Model Building: Data scientists collaborate with team members to deploy models and transform raw data into a format usable for further data science.

Data Engineers' Role: Skilled in software engineering, data engineers build scalable data pipelines integrating models with systems like web applications or Apache Kafka.

Who uses Spark? Data Engineering

Use of Spark in Data Engineering: Apache Spark's Structured Streaming APIs aid engineers in building pipelines for real-time and static data, simplifying parallel computations and data distribution.

Advancements in Spark for Enhanced Engineering: Updates in Spark 2.x and 3.0, like the Catalyst optimizer and Tungsten, have improved data engineering by offering efficient code generation and a choice of APIs—RDDs, DataFrames, or Datasets

Popular Spark use cases

Whether you are a data engineer, data scientist, or machine learning engineer, you'll find Spark useful for the following use cases:

- Processing in parallel large data sets distributed across a cluster
- Performing ad hoc or interactive queries to explore and visualize data sets
- Building, training, and evaluating machine learning models using MLlib
- Implementing end-to-end data pipelines from myriad streams of data
- Analyzing graph data sets and social networks

SEAS 8515

Next Class:

Apache Spark's Structured APIs, Spark SQL and DataFrames: Introduction to Built-in Data Sources

School of Engineering & Applied Science
THE GEORGE WASHINGTON UNIVERSITY

Dr. Adewale Akinfaderin