Introduction to Spark

DSCI-D 351 Big Data Analytics

Yuhui Hong

Luddy School of Informatics, Computing, and Engineering
Indiana University Bloomington

September 3, 2024

Overview

What is Big Data?

Big data primarily refers to data sets that are too large or complex to be dealt with by traditional data-processing application software.

from *Wikipedia*

Challenges of Big Data Processing

- Scalability: Traditional systems struggle with horizontal scaling for large data volumes.
- Speed: Processing large datasets can be slow, especially for real-time analysis.
- Complexity: Big Data often comes in various formats (structured, unstructured, semi-structured), which traditional systems are not designed to handle efficiently.

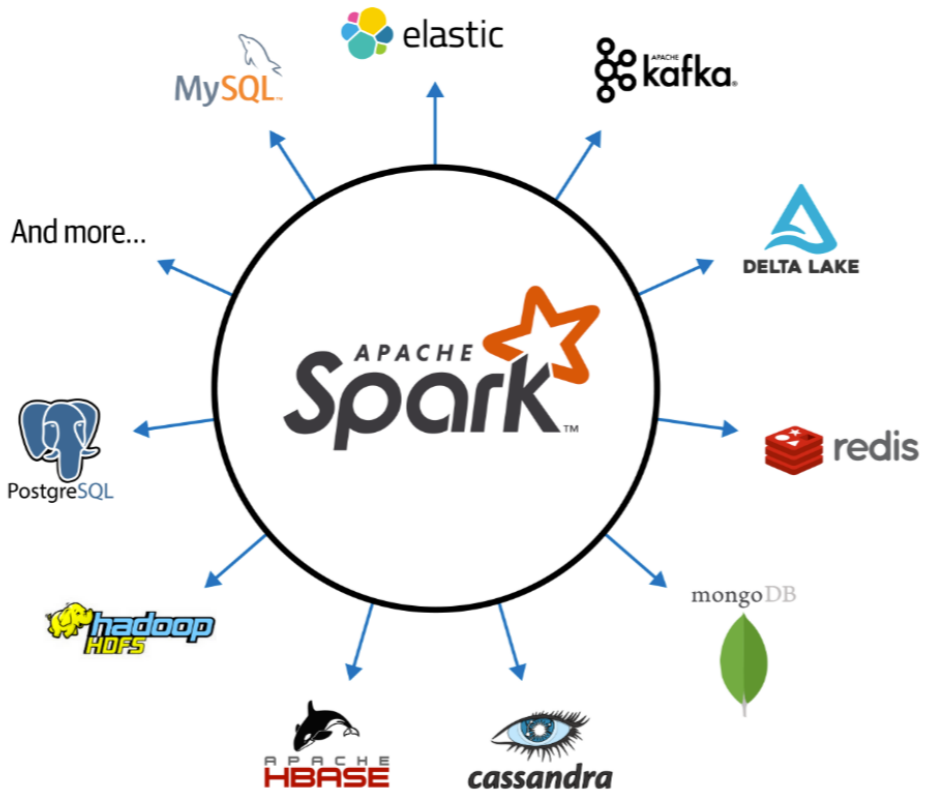
Distributed Data Processing

In distributed data processing, tasks on large-scale data are broken down into smaller units that can be processed in parallel. Popular distributed computing frameworks include Apache Hadoop, Apache Spark, Google BigQuery, Apache Flink, Dask, etc.

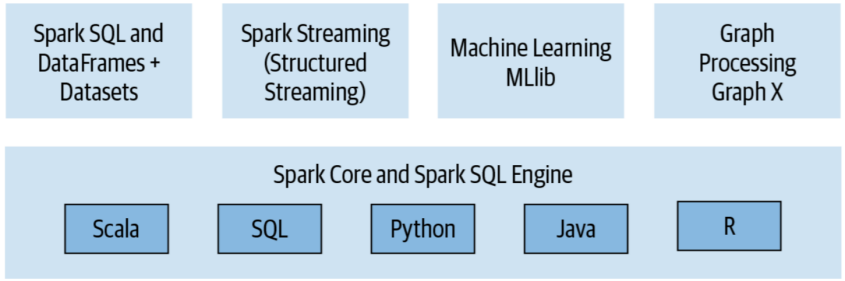
Brief History of Apache Spark

1. Origins at UC Berkeley (2009-2010): Spark was developed in 2009 by a team at UC Berkeley's AMPLab, led by Matei Zaharia. It was designed as a faster alternative to Hadoop's MapReduce, focusing on in-memory processing for improved speed in iterative tasks.
2. Open Sourcing and ASF Project (2010-2013): Spark was open-sourced in 2010 and became an Apache Software Foundation (ASF) project in 2013. This transition accelerated its development and adoption in the big data community.
3. Rapid Growth and Ecosystem Expansion (2014-2015): By 2014, Spark became the most active project in the Apache community. During this time, key components like Spark SQL, Spark Streaming, MLlib, and GraphX were introduced, expanding its capabilities.
4. Widespread Industry Adoption (2015-Present): Spark saw rapid adoption across industries, supported by companies like Databricks, IBM, and Cloudera. It has since become a leading tool for big data analytics, known for its speed, flexibility, and robust ecosystem.

Apache Spark's ecosystem of connectors

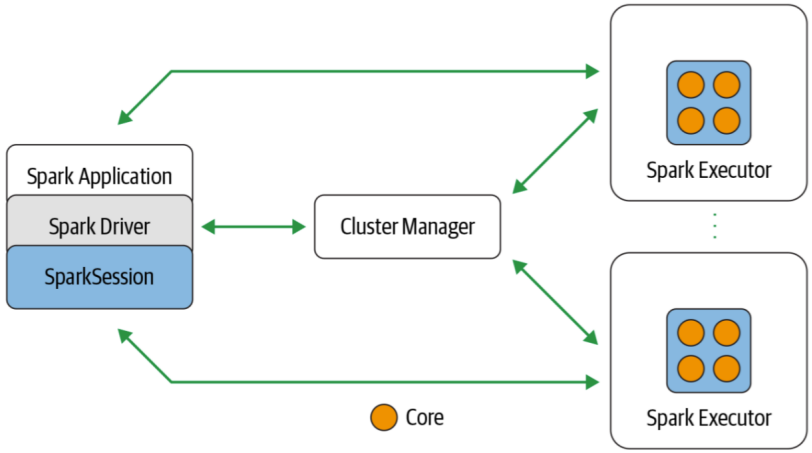


Apache Spark Core and API Stack



Spark Core is the foundation of Apache Spark. It is responsible for memory management, fault recovery, scheduling, distributing and monitoring jobs, and interacting with storage systems.
Spark offers four distinct components as libraries for diverse workloads: Spark SQL, Spark Structured Streaming, Spark MLlib, and GraphX.

Overview



(1) **Spark driver** breaks down the application into tasks, requests resources from the **cluster manager**, and distributes tasks to **executors**.
(2) **Cluster manager** allocates resources and informs **executors**.
(3) **Spark executors** execute the tasks and perform computations on data, sending results back to the **driver**.

Spark Driver, Cluster Manager, and Spark Excutor

- **Spark driver**: The Spark driver, which initializes the `SparkSession`, handles multiple tasks: it interacts with the cluster manager to request resources (CPU, memory, etc.), converts Spark operations into DAG computations, schedules them, and distributes tasks to executors. Once resources are allocated, it directly communicates with the executors.
- **Cluster Manager**: The cluster manager allocates resources for the Spark application's nodes. Spark supports four cluster managers: local, standalone, YARN (client), YARN (cluster), and Kubernetes.
- **Spark Executor**: A Spark executor runs on each worker node, executing tasks and communicating with the driver program. Typically, only one executor per node is used.

'SparkSession'

- **SparkSession**: Spark contexts (e.g., `SparkContext`, `SQLContext`, `HiveContext`, `StreamingContext`) provide specialized functionalities for different types of data processing,

  while `SparkSession` unifies these functionalities into a single entry point for managing Spark applications and querying structured and unstructured data in Spark 2.0 and later.

Create and Close a 'SparkSession'

In Python:

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder \
        .appName("Example") \
        .getOrCreate()

# Load and process data

# Stop the SparkSession
spark.stop()
```

In Scala:

```scala
import org.apache.spark.sql.SparkSession

// Create a SparkSession
val spark = SparkSession.builder
          .appName("Example")
          .getOrCreate()

// Load and process data

// Stop the SparkSession
spark.stop()
```

RDDs and DAG

Resilient Distributed Datasets (RDDs) is Spark's core data structure, representing distributed, immutable collections of data that can be processed in parallel. RDDs support:

- Transformations: Lazy operations like map and filter that define a new RDD.
- Actions: Operations like collect and count that trigger computation and return results.

Directed Acyclic Graph (DAG) represents the sequence of transformations and actions applied to RDDs. Spark builds a DAG for each job, optimizing the execution plan and managing dependencies to ensure efficient and fault-tolerant processing.

Transformations and Actions on RDDs

```python
# Create an RDD
rdd = spark.sparkContext.parallelize([1, 2, 3, 4])

# Perform transformations
rdd2 = rdd.map(lambda x: x * 2)  # Multiply each element by 2
rdd3 = rdd2.filter(lambda x: x > 4)  # Filter elements greater than 4

# Perform action
result = rdd3.collect()  # Collect results to the driver

# Print results
print(result)  # Output will be [6, 8]
```

In this example, Spark doesn't start processing the data until `collect()` is called. Once an action is triggered, Spark optimizes and executes the chain of transformations to produce the final result.

DAG of Above Transformations and Actions

```
[1, 2, 3, 4]  # Initial RDD: rdd
    |
    | map(lambda x: x * 2)
    v
 [2, 4, 6, 8]  # Transformed RDD: rdd2
    |
    | filter(lambda x: x > 4)
    v
 [6, 8]  # Filtered RDD: rdd3
    |
    | collect()
    v
 [6, 8]  # Result: result
```

DataFrames, Graphs, etc.

In addition to RDDs, Spark supports various other data structures through its specialized libraries, such as DataFrames via `Spark SQL`, Graphs via `GraphX`, and machine learning models via `MLlib` [2].

Transformations and Actions on DataFrame

```python
# Create a DataFrame
data = [("Alice", 30), ("Bob", 25), ("Cathy", 28)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns)

# Show the original DataFrame
print("Original DataFrame:")
df.show() # This is an action on DataFrame.
```

```
Original DataFrame:
+-----+---+
| Name|Age|
+-----+---+
|Alice| 30|
|  Bob| 25|
|Cathy| 28|
+-----+---+
```

Transformations and Actions on DataFrame (cont.)

```python
from pyspark.sql.functions import col

# Perform a transformation: filter rows where age is greater than 25
df_filtered = df.filter(col("Age") > 25)

# Perform another transformation: add a new column with age incremented by 1
df_transformed = df_filtered.withColumn("Age Plus One", col("Age") + 1)

# Show the transformed DataFrame
print("Transformed DataFrame:")
df_transformed.show()
```

```
Transformed DataFrame:
+-----+---+-----------+
| Name|Age|Age Plus One|
+-----+---+-----------+
|Alice| 30|         31|
|Cathy| 28|         29|
+-----+---+-----------+
```

Finding Maximum Temperatures by City

Assume you have five files and each file contains two columns. These columns represent a key and a value in Hadoop terms that represent a city and the corresponding temperature recorded in that city for the various measurement days.

For example, the first file contains:

```
Toronto, 20
Whitby, 25
New York, 22
Rome, 33
```

Implement MapReduce Algorithm: Initialization

Let's implement it based on RDDs. So, we first need to initialize a SparkContext and load data to it.

```
from pyspark import SparkContext

# Initialize Spark Context
sc = SparkContext(master="local", appName="TemperatureMax")

# Read the data from text files
data = sc.textFile("temperature_data/*.txt")
```

Implement MapReduce Algorithm: Map Phase

```
# Map phase: Parse each line into a city and temperature tuple
def parse_line(line):
    city, temp = line.split(',')
    return city.strip(), int(temp.strip())

city_temps = data.map(parse_line)
```

If you apply `collect()` to `city_temps`, you will see that `city_temps` contains:

```
[
    ('Toronto', 20),
    ('Whitby', 25),
    ('New York', 22),
    ('Rome', 33),
    ('Toronto', 18),
    ('Whitby', 27),
    ......
]
```

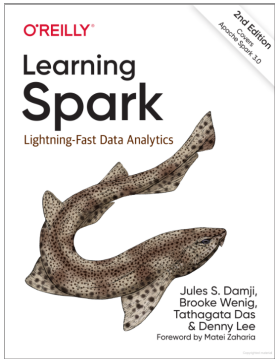
Implement MapReduce Algorithm: Reduce Phase

```
# Reduce phase: Reduce by key to find the maximum temperature for each city
max_temps_by_city = city_temps.reduceByKey(lambda x, y: max(x, y))

# Collect and display the results
results = max_temps_by_city.collect()

for city, max_temp in results:
    print(f"{city}: {max_temp}")
```

The output will look like this:

```
Toronto: 32
Whitby: 27
New York: 33
Rome: 38
```

References



Jules, TD Damji, Brooke Wenig, and D. Lee. *Learning Spark: Lightning-Fast Data Analytics*. O'Reilly Media, Inc., 2020.

Apache Spark. *Python API Documentation*. 2024. Available at: `https://spark.apache.org/docs/latest/api/python/index.html` (Accessed: 2024-08-31).

Take Away

- Components and architecture of Spark
  - Spark Driver
  - Cluster Manager
  - Spark Executors
- Two core abstracts in Spark: RDDs and DAG
- Using Spark to implement a simple MapReduce algorithm

Thank you!