Programming Hadoop with Spark

Bernard Lee Kok Bang

What is Spark?

- "multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters" - *spark.apache.org*

- Allows more complications tasks like machine learning, data mining, graph analysis, and streaming data **[vs. pig or hive]**

Spark is scalable

It's Scalable

Driver Program -Spark Context → Cluster Manager (Spark, YARN) → Executor - Cache -Tasks
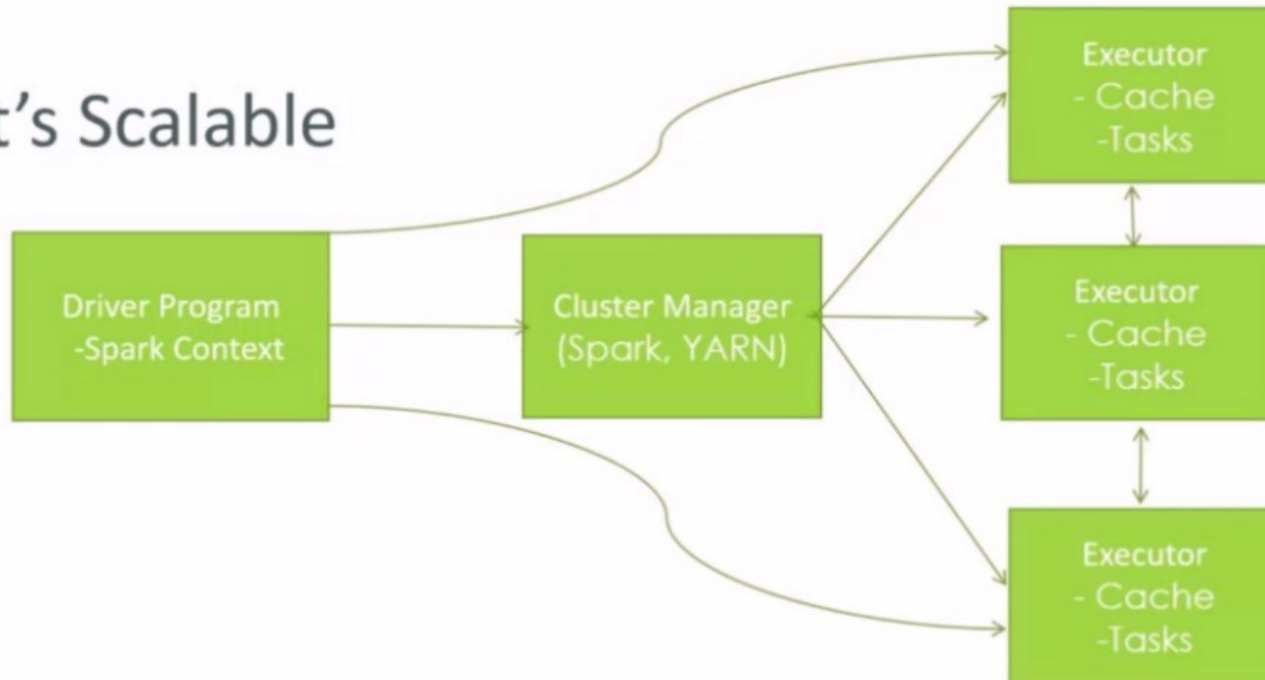
[unlike disk-based solution]

- A **driver program** with script that control what's going to happen with the job
- **Cluster manager** will distribute the job across entire cluster of commodity computers
  - →Process all the data in parallel
- Each **executor** has cache and responsible tasks
  - →**Cache** is the key to performance
- Spark is a **memory-based solution**, tries to retain as much information in RAM as possible
  - →Important key to its **speed**

It's fast

- "Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk"

- DAG engine (directed acyclic graph) optimizes workflows
  - →Work backwards from the end result and figure out the fastest way to get to the intended goal

- Code written in Python, Java, or Scala

- Built around one main concept: the *Resilient Distributed Dataset (RDD)*
  - →An *object* that represents a dataset
  - →*functions* can be called on the RDD object to *transform, reduce, or analyze* to produce new RDDs

Key Characteristics of RDD

1. *Resilient*: Automatic fault recovery
   - using the lineage information *(the sequence of transformations)* recorded during its creation

2. *Distributed*: Parallel processing across multiple nodes in a cluster
   - Each partition of an RDD is *processed independently* on different nodes

3. *Immutable*: Content cannot be changed
   - can apply transformations to RDDs to derive new RDDs, but the original RDD remains unchanged
   - immutability *simplifies fault tolerance and facilitates parallel processing*

4. Lazy Evaluation: Delayed computation for optimization
   - transformations on RDDs are *not computed immediately*

5. In-Memory Computation: Fast data processing with memory storage
   - enables fast data processing by *minimizing disk I/O operations*
   - efficient for *iterative and interactive* data processing tasks

RDDs vs DataFrames

Feature	RDD	DataFrame
Level of Abstraction	Low-level (closer to raw data)	High-level (like SQL tables)
Ease of Use	Requires more code (functional programming)	Easier with SQL-like syntax
Performance	Slower due to no optimization	Faster due to Catalyst Optimizer
Optimization	No built-in optimization	Optimized with Catalyst and Tungsten
Data Structure	No schema (just a collection of objects)	Schema-based (columns with types)

RDD

```
rdd = sc.parallelize([("John", 28), ("Jane", 35)])
filtered = rdd.filter(lambda x: x[1] > 30)
print(filtered.collect())
# Output: [('Jane', 35)]
```

*No schema
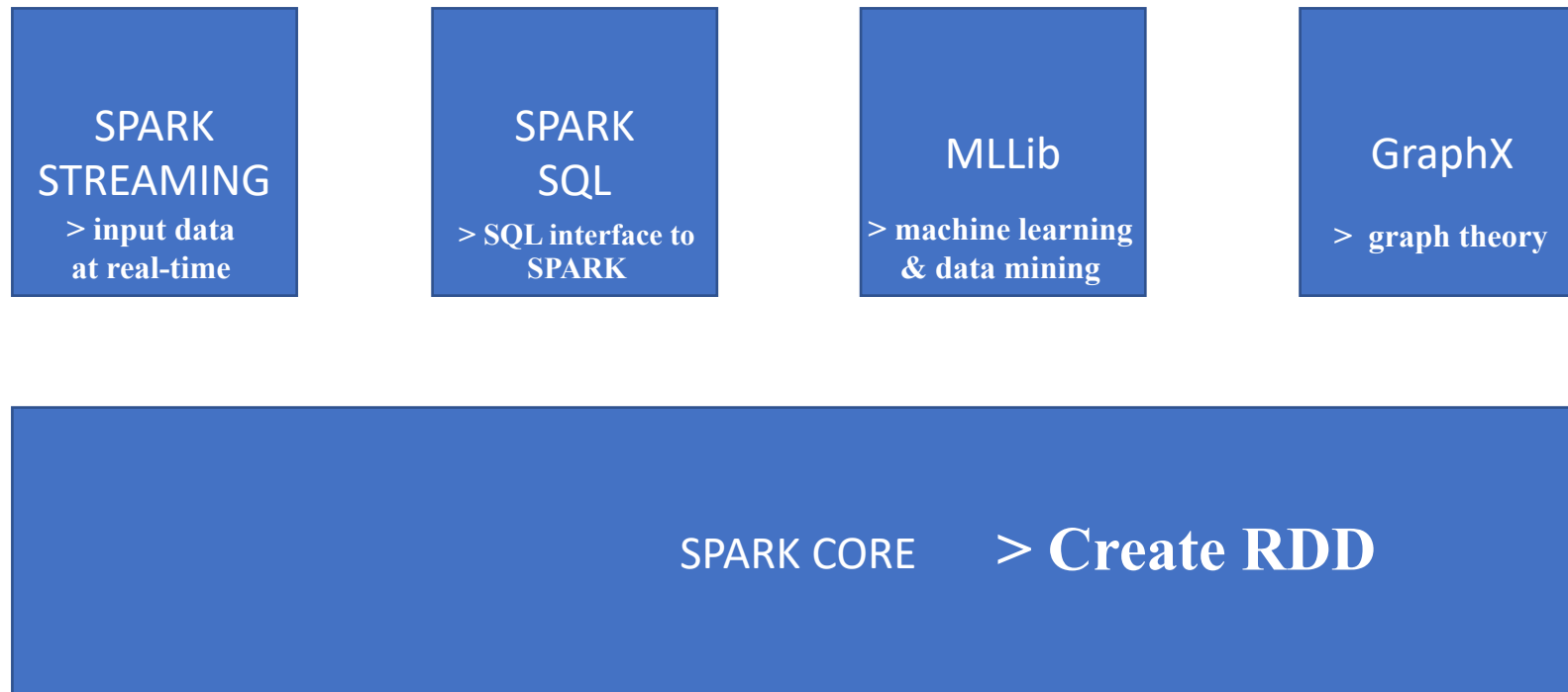(just a collection of objects)*

DataFrame

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
df = spark.createDataFrame([("John", 28), ("Jane", 35)], ["name", "age"])
filtered_df = df.filter(df.age > 30)
filtered_df.show()
# Output:
# +----+---+
# |name|age|
# +----+---+
# |Jane| 35|
# +----+---+
```

*Schema-based
(columns with types)*

Spark ecosystem

The SparkContext - running environment within Spark

- Created by the driver program **> create a Spark context [environment]**
- Responsible for making RDD resilient and distributed
- The Spark shell creates a **"sc"** object

Understanding SparkContext

- SparkContext → the entry point to Apache Spark
- Core Functionality
  1. *Initialization*: Establishes connection to Spark execution environment
  2. *Data Operations*: Create RDDs, perform transformations, and actions
  3. *Resource Management*: Coordinates computing resources for Spark application
  4. *Central Control*: Manages all Spark-related operations and resources

Example of Creating RDD code

Takes a list of numbers, distributes it across multiple nodes in a computing cluster
> *nums = parallelize([1, 2, 3, 4])*

Using the SparkContext (sc) to read a file
triple forward slashes ("///") indicate files on the local filesystem
double forward slashes ("//") indicate otherwise: s3n://, hdfs://
> *sc.textFile("file:///c:/users/xxx/xxx.txt")*

Create a HiveContext within the Spark environment
> *hiveCtx = HiveContext(sc)*
> *rows = hiveCtx.sql("SELECT name, age FROM users")*

Admin password setting using puTTY

- change settings in Ambari for running Spark

```
1  su root
2  Password:
3  ambari-admin-password-reset
4  Please set the password for admin:
5  Please retype the password for admin:
```

password: stqd6324

Change Configuration in Spark

1. Log in to Ambari as **admin**

2. Go to **Services** and choose **Spark2** and **Config** tab

3. Click **Advanced spark2-log4j-properties**

4. Change "log4j.rootCategory=INFO, console" to "**log4j.rootCategory=ERROR, console**"

5. Click **Save**

6. Save configuration as "**Change log level to ERROR in Spark2**", click Save   **Click 'Proceed anyway'**

7. Restart Spark2 **Restart all affected; Confirm Restart All**

Advanced livy2-log4j-properties

```
# Set everything to be logged to the console
log4j.rootCategory=ERROR, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n

log4j.logger.org.eclipse.jetty=WARN
```

Transforming RDD

- map **[for one-to-one relationship]**
- flatmap **[for different number of rows in the output]**
- filter
- distinct
- sample
- union, intersection, substract

map example

- rdd = sc.parallelize([1, 2, 3, 4])
- squaredRDD = rdd.map(lambda x:x*x)

- **lambda function: take each input row from RDD, call it x, and return the value x times x**

- This yields 1, 4, 9, 16

Hands on using map() function

```python
from pyspark import SparkContext

# Initialize SparkContext
sc = SparkContext("local", "PrintSquaredRDD")

# Create an RDD containing a list of numbers
rdd = sc.parallelize([1, 2, 3, 4])

# Apply map transformation to square each element
squaredRDD = rdd.map(lambda x: x * x)

# Collect the result into a list and print it
result = squaredRDD.collect()
print("Squared RDD Result:", result)

# Stop SparkContext
sc.stop()
```

spark-submit yourScript.py Optionally, can run use python <file path>
e.g., python hazim/Script1.py

What's that lambda thing?

- Many RDD methods accept a *function* as a parameter
  - *rdd.map(lambda x: x*x)*   **> one liner function**

    * functional programming

- Is the same thing as

```
def squareIt(x)
    return x*x
rdd.map(squareIt)
```

This is Python!!!

Hands on using **flatmap()** function

vi yourScript.py

```python
# Import SparkContext from the PySpark library
from pyspark import SparkContext

# Initialize SparkContext
# Connects to a Spark cluster running locally on the same machine
# Gives the Spark application the name "FlatMapExample"
sc = SparkContext("local", "FlatMapExample")

# Create an RDD containing a list of words
words_rdd = sc.parallelize(["Hello", "world", "how", "are", "you"])

# Define a function to split each word into its individual characters
def split_word(word):
    return list(word) # list() splits the string into its individual characters

# Apply flatMap transformation to split each word into individual characters
characters_rdd = words_rdd.flatMap(split_word)

# Collect the result into a list for printing
result = characters_rdd.collect()

# Print the result
print("Original Words:", words_rdd.collect())
print("Characters:", result)
```

spark-submit yourScript.py

Lazy evaluation

- Nothing happens in the driver program until an action is called!
- Spark will *work backwards*
- Figure out the *fastest way* to achieve the result we want!!
- Building *chain of dependency graph* within our driver script

Spark SQL

Resilient Distributed Dataset

- Spark2 - Dataframe and Datasets
- Extends RDD to a "Dataframe" object
- **DataFrames**:

  - Contain Row objects

  - Can run SQL queries

  - Has a schema (leading to more efficient storage)

need to define data type and field name in advance

Using SparkSQL in Python

- *from pyspark.sql import SQLContext, Row*

- *hiveContext = HiveContext(sc)*

- *myResultDataFrame = hiveContext.sql("""SELECT xxx FROM xxx ORDER BY xxx""")*

- *myResultDataFrame.select("someFieldName")*

- ***myResultDataFrame.filter("someFieldName > 10")***

- *myResultDataFrame.groupBy("someFieldName").mean()*

Let's start playing with Spark2 using terminal

- Find the worst average rating movies from movielens datasets using Spark SQL

1. Set the Spark SQL running environment

```python
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.sql import functions
```

2. Define the function

```python
def loadMovieNames():
    movieNames = {}
    with open("ml-100k/u.item") as f:        > make sure you have the 'u.item'
        for line in f:                             in ml-100k folder in your 'local'
            fields = line.split('|')
            movieNames[int(fields[0])] = fields[1]
    return movieNames           movieID              movieTitle

def parseInput(line):
    fields = line.split()
    return Row(movieID = int(fields[1]), rating = float(fields[2]))
```

3. Spark SQL

> getOrCreate() is a method provided by the SparkSession class in PySpark that allows us to either create a new SparkSession or get an existing one, which is necessary for accessing the Dataset and DataFrame API in Spark.

```python
if __name__ == "__main__":
    # Create a SparkSession
    spark = SparkSession.builder.appName("PopularMovies").getOrCreate()

    # Load up our movie ID -> name dictionary
    movieNames = loadMovieNames()

    # Get the raw data
    lines = spark.sparkContext.textFile("hdfs:///user/maria_dev/ml-100k/u.data")
    # Convert it to a RDD of Row objects with (movieID, rating)
    movies = lines.map(parseInput)
    # Convert that to a DataFrame
    movieDataset = spark.createDataFrame(movies)

    # Compute average rating for each movieID
    averageRatings = movieDataset.groupBy("movieID").avg("rating")

    # Compute count of ratings for each movieID
    counts = movieDataset.groupBy("movieID").count()
```

This is from HDFS

Activate Windows

Example output of the movieNames dictionary

```python
{
  1: 'Toy Story (1995)',
  2: 'GoldenEye (1995)',
  3: 'Four Rooms (1995)',
  4: 'Get Shorty (1995)',
  5: 'Copycat (1995)',
  ...
  1682: 'Scream of Stone (Schrei aus Stein) (1991)',
  1683: 'Jupiter's Wife (1995)',
  1684: 'Tom and Huck (1995)',
  1685: 'Show, The (1995)',
  1686: 'Robin Hood: Men in Tights (1993)',
  ...
}
```

3. Spark SQL (cont...)

```
# Join the two together (We now have movieID, avg(rating), and count columns)
averagesAndCounts = counts.join(averageRatings, "movieID")

# Pull the top 10 results
topTen = averagesAndCounts.orderBy("avg(rating)").take(10)

# Print them out, converting movie ID's to names as we go.
for movie in topTen:
    print (movieNames[movie[0]], movie[1], movie[2])

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

movieID   **count**   **avg(rating)**

movieNames from 'local'

4. Submit the python script from terminal

```
spark-submit yourScript.py
```

Output: Worst average rating movies

```
('Further Gesture, A (1996)', 1, 1.0)
('Falling in Love Again (1980)', 2, 1.0)
('Amityville: Dollhouse (1996)', 3, 1.0)
('Power 98 (1995)', 1, 1.0)
('Low Life, The (1994)', 1, 1.0)
('Careful (1992)', 1, 1.0)
('Lotto Land (1995)', 1, 1.0)
('Hostile Intentions (1994)', 1, 1.0)
('Amityville: A New Generation (1993)', 5, 1.0)
('Touki Bouki (Journey of the Hyena) (1973)', 1, 1.0)
```

Bonus

- **Movie recommendations using MLLib**

First do this

change the u.data file name to u_edited.data

Fabricate the u.data in HDFS

				action movies
0	50	5	881250949	Star Wars
0	172	5	881250949	The Empire Strikes Back
0	133	1	881250949	Gone with the Wind
196	242	3	881250949	
186	302	3	891717742	
22	377	1	878887116	romance movie
244	51	2	880606923	

Second do this

sudo pip install numpy==1.16

1. Set the Spark SQL running environment

```python
from pyspark.sql import SparkSession
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row
from pyspark.sql.functions import lit
```

2. Define the function

```python
# Load up movie ID -> movie name dictionary
def loadMovieNames():
    movieNames = {}
    with open("ml-100k/u.item") as f:
        for line in f:
            fields = line.split('|')
            movieNames[int(fields[0])] = fields[1].decode('ascii', 'ignore')
    return movieNames

# Convert u.data lines into (userID, movieID, rating) rows
def parseInput(line):
    fields = line.value.split()
    return Row(userID = int(fields[0]), movieID = int(fields[1]), rating = float(fields[2]))
```

3. Spark MLLib

```python
if __name__ == "__main__":
    # Create a SparkSession (the config bit is only for Windows!)
    spark = SparkSession.builder.appName("MovieRecs").getOrCreate()

    # This line is necessary on HDP 2.6.5:
    spark.conf.set("spark.sql.crossJoin.enabled", "true")

    # Load up our movie ID -> name dictionary
    movieNames = loadMovieNames()

    # Get the raw data
    lines = spark.read.text("hdfs:///user/maria_dev/ml-100k/u.data").rdd

    # Convert it to a RDD of Row objects with (userID, movieID, rating)
    ratingsRDD = lines.map(parseInput)

    # Convert to a DataFrame and cache it
    ratings = spark.createDataFrame(ratingsRDD).cache()

    # Create an ALS collaborative filtering model from the complete data set
    als = ALS(maxIter=5, regParam=0.01, userCol="userID", itemCol="movieID", ratingCol="rating")
    model = als.fit(ratings)
```

change to u_edited.data

train the model using ratings data frame

3. Spark MLLib (cont...)

```python
# Print out ratings from user 0:
print("\nRatings for user ID 0:")
userRatings = ratings.filter("userID = 0")
for rating in userRatings.collect():
    print movieNames[rating['movieID']], rating['rating']

print("\nTop 20 recommendations:")
# Find movies rated more than 100 times
ratingCounts = ratings.groupBy("movieID").count().filter("count > 100")
# Construct a "test" dataframe for user 0 with every movie rated more than 100 times
popularMovies = ratingCounts.select("movieID").withColumn('userID', lit(0))

# Run our model on that list of popular movies for user ID 0
recommendations = model.transform(popularMovies)

# Get the top 20 movies with the highest predicted rating for this user
topRecommendations = recommendations.sort(recommendations.prediction.desc()).take(20)

for recommendation in topRecommendations:
    print (movieNames[recommendation['movieID']], recommendation['prediction'])

spark.stop()
```

retrieve all the rows at once as a list

adds a new column to this DataFrame with the name userID and the value 0 for every row.

4. Submit the python script from terminal

Top 20 movies recommendation for UserID 0

```
Top 20 recommendations:
(u'Wrong Trousers, The (1993)', 5.749821662902832)
(u'Fifth Element, The (1997)', 5.2325282096862793)
(u'Close Shave, A (1995)', 5.0506253242492676)
(u'Monty Python and the Holy Grail (1974)', 4.9965953826904297)
(u'Star Wars (1977)', 4.9895496368408203)
(u'Army of Darkness (1993)', 4.980320930480957)
(u'Empire Strikes Back, The (1980)', 4.9729299545288086)
(u'Princess Bride, The (1987)', 4.9577054977416992)
(u'Blade Runner (1982)', 4.9106745719909668)
(u'Return of the Jedi (1983)', 4.7780814170837402)
(u'Rumble in the Bronx (1995)', 4.6917591094970703)
(u'Raiders of the Lost Ark (1981)', 4.6367182731628418)
(u"Jackie Chan's First Strike (1996)", 4.632108211517334)
(u'Twelve Monkeys (1995)', 4.6148405075073242)
(u'Spawn (1997)', 4.5741710662841797)
(u'Terminator, The (1984)', 4.5611510276794434)
(u'Alien (1979)', 4.5415172576904297)
(u'Terminator 2: Judgment Day (1991)', 4.529487133026123)
(u'Usual Suspects, The (1995)', 4.5179119110107422)
(u'Mystery Science Theater 3000: The Movie (1996)', 4.5095906257629395)
```

User ID 0 : likes action movies [rating = 5]
: doesn't like romance movies [rating = 1]