Introduction to Spark and GeoSpark

Yijun Lin
Department of Computer Science & Engineering
University of Minnesota, Twin Cities
lin00786@umn.edu
Hadoop MapReduce VS. Apache Spark

• Hadoop MapReduce
  • Typically, data are read from disk, processed, and written back to disk
  • MapReduce is inefficient for multi-pass applications that read data more than once

• Apache Spark
  • When the output of an operation needs to be fed into another operation, Spark passes the data directly without writing to persistent storage
What is Spark

• Apache Spark is an open-source cluster computing framework

• Application areas
  • Iterative Algorithms
  • Interactive Data Mining
  • Streaming Applications

Spark Stack

- Spark Libraries
  - Spark SQL & DataFrames
  - MLlib
  - GraphX
  - Spark Streaming

- Core Module
  - Spark Core

- JobSchedulers
  - YARN
  - Mesos
  - Standalone

- Data Storage Layer
  - Hadoop HDFS
  - Cassandra
  - HBase
  - S3

* Resilient Distributed Datasets (RDDs)
Resilient Distributed Datasets (RDDs)

- An RDD is an **immutable, in-memory** collection of objects.
- Each RDD can be split into multiple partitions, which in turn are computed on different nodes of the cluster, so that users can
  - Explicitly persist intermediate results in memory
  - Control the partitioning to optimize data operations
  - Manipulate data using a rich set of operators

- RDDs seem a lot like Scala collections
  - RDD[T] and List[T]
Partitioning Strategy of RDDs

• Spark partitioning
  • Dividing the data into chunks that consider the number of partitions (cluster size) and how data is distributed across partitions.

• Number of partitions? Cannot be too large or too small

• How are data distributed across partitions?
  • HashPartitioner will distribute data, e.g., (key, value) pairs, across the partitions using
    \[ partitionId = hash(Key) \mod n\_partitions \]
  • RangePartitioner will distribute data across partitions based on a specific range
  • Customized Partitioner

```python
def partition_func(x):
    # x: (word, 1)
    # ord("a") => 97 \% n\_partitions
    # ord("p") => 80 \% n\_partitions
    # ord("Z") => 90 \% n\_partitions
    return ord(x[0][0])

word_rdd = word_rdd.partitionBy(n\_partitions, partition_func)
```
RDD Operations - Transformation VS. Action

• Transformation
  • Return new RDDs as results
  • They are lazy, the result RDD is not immediately computed

```
def length_rdd(rdd):
  return rdd.map(lambda x: len(x))
```

• Action
  • Compute a result based on an RDD, and returned
  • They are eager, the result is immediately computed

```
coll = rdd.collect()
print(coll)  # ['you', 'jump', 'I', 'jump', '']
```
Example

- Consider the following example:

```python
a_list = ['you', 'jump', 'I', 'jump', '']
# create an RDD from a list
a_rdd = sc.parallelize(a_list)  # RDD[String]
# call a map operation RDD
a_len_rdd = a_rdd.map(lambda x: len(x))  # RDD[Int]
```

What has happened on the cluster at this point?

sc - a SparkContext (or SparkSession) object
The SparkContext object can be thought as your handle to the Spark cluster. It represents the connection between the Spark cluster and your running application. Initializing a SparkContext or SparkSession object is the first step of a Spark program.
Example

• Consider the following example:

```python
a_list = ['you', 'jump', 'I', 'jump', '']
# create an RDD from a list
a_rdd = sc.parallelize(a_list)  # RDD[String]
# call a map operation RDD
a_len_rdd = a_rdd.map(lambda x: len(x))  # RDD[Int]
```

What has happened on the cluster at this point?

**Nothing.** Execution of map (a transformation) is deferred.
Example (Cont.)

• Consider the following example:

```scala
a_list = ['you', 'jump', 'I', 'jump', '']
# create an RDD from a list
a_rdd = sc.parallelize(a_list)  # RDD[String]
# call a map operation RDD
a_len_rdd = a_rdd.map(lambda x: len(x))  # RDD[Int]
total_len = a_len_rdd.reduce(lambda a, b: a + b)  # 12
```

`reduce(op: (A, A) => A): A`
Combine the elements in the RDD together using op function and return result

Spark starts the execution when an action is called
Return the total number of characters in the entire RDD of strings
Benefits of Laziness

• Another example:

```python
input_file = 'work-count-sample-doc.txt'
text_rdd = sc.textFile(input_file)
word_rdd = text_rdd.flatMap(lambda x: x.split(' ')).take(10)
```

• The execution of `flatMap` is **deferred** until `take` action happens
  • As soon as the first 10 elements of have been computed, `word_rdd` is done

• Spark analyzes and optimizes the **chain of operations** before executing it
  • Spark saves time and space by avoiding unnecessary computation
Common Transformations

map\( \text{map}(f: A \rightarrow B): \text{RDD}[T] \)
Apply function to each element in the RDD and return an RDD of the result.

flatmap\( \text{flatmap}(f: A \rightarrow B): \text{RDD}[T] \)
Apply function to each element in the RDD and return an RDD of the result, but output is flattened.

filter\( \text{filter}(\text{pred}: A \rightarrow \text{Boolean}): \text{RDD}[T] \)
Apply predicate function, \text{pred}, to each element in the RDD and return an RDD of elements that passed the condition.

distinct\( \text{distinct}(): \text{RDD}[T] \)
Return an RDD with duplicates removed
Common Transformations

**flatmap**  
`flatmap[T](f: A=>B): RDD[T]`  
Apply function to each element in the RDD and return an RDD of the result, but output is flattened.

```scala
val text: List[String] = List("you and me", "jump and run", "I love you", "jump forward", ")
val textRDD = sc.parallelize(text)

val splitText = textRDD.flatMap(phase => phase.split(" ")) // Flatten the output

val splitTextColl = splitText.collect()
splitTextColl.foreach(println) // "you", "me", "jump", "and", "run", "I", "love", "you", "jump", "forward"
```
**Common Transformations**

**distinct**  
Distinct: `distinct(): RDD[T]`  
Return an RDD with duplicates removed

```scala
val text: List[String] = List("you and me", "jump and run", "I love you", "jump forward","")
val textRDD = sc.parallelize(text)

val splitText = textRDD.flatMap(phase => phase.split(" "))  // Flatten the output
val textDist = splitText.distinct()  // Get the distinct words

val textDistColl = textDist.collect()
textDistColl.foreach(println)  // "me", "I", "love", "run", "forward", "jump", "you", "and"
```
Common Actions

**collect**  
**collect: Array[T]**  
Return all elements from RDD.

**count**  
**count(): Long**  
Return the number of elements in the RDD.

**take**  
**take(num: Int): Array[T]**  
Return the first num elements of the RDD.

**reduce**  
**reduce(op: (A, A) => A): A**  
Combine the elements in the RDD together using op function and return result.

**foreach**  
**foreach(f: A => Unit): Unit**  
Apply function to each element in the RDD, and return Unit.
Common Actions

**count**  
**count(): Long**  
Return the number of elements in the RDD.

```scala
val text: List[String] = List("you and me", "jump and run", "I love you", "jump forward", "")
val textRDD = sc.parallelize(text)

val splitText = textRDD.flatMap(phase => phase.split(" ")) // Flatten the output
val textDist = splitText.distinct() // Get the distinct words
val counts = textDist.count() // return 8
```
How Spark Jobs are Executed
How Spark Jobs are Executed

This is the node you’re interacting with when you’re writing Spark programs!

These are the nodes actually executing the jobs!
How Spark Jobs are Executed

Driver Program
Spark Context

Cluster Manager

via a cluster manager.
Allocates resources across cluster, manages scheduling.
e.g., YARN/Mesos

Worker Node
Executor

Worker Node
Executor

Worker Node
Executor
How Spark Jobs are Executed

Steps of executing a Spark program:
1. The driver program runs the Spark application, which creates a SparkContext
2. The SparkContext connects to a cluster manager to allocate resources
3. Spark acquires executors on nodes in the cluster, which run computations for your application.
4. Driver program sends your application code to executors to execute.
Example

- A simple example with `println` (Scala code)

```scala
case class Person(name: String, age: Int)
val people: RDD[Person]
people.foreach(println)
```

foreach(f: A => Unit): Unit
Apply function to each element in the RDD and return Unit

What will you see?
Example

• A simple example with `println` (Scala code)

```scala
case class Person(name: String, age: Int)
val people: RDD[Person]
people.foreach(println)
```

`foreach(f: A => Unit): Unit`  
Apply function to each element in the RDD and return Unit

On the driver: Nothing.

The operation `foreach` is an action, with return type `Unit`. Therefore, it is eagerly executed on the executors, not the driver. Thus, `println` are happening on the worker nodes and return nothing to the driver node.
Programming with Spark

• Example: Logistic Regression
  • Logistic regression is an iterative algorithm typically used for classification. Like most classification algorithms, it updates weights iteratively based on the training data.

\[ w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w, x_i, y_i) \]
Programming with Spark

• Example: Logistic Regression
  - Logistic regression is an iterative algorithm typically used for classification. Like most classification algorithms, it updates weights iteratively based on the training data.

```scala
val points = sc.textFile(...).map(parsePoint) // case class Point(x: Double, y: Double)
var w = Vector.zeros(d)
for(i <- 1 to numIterations) {
    val gradient = points.map { p =>
        g(p) // Apply the function of logistic regression
    }.reduce(_ + _)
    w -= alpha * gradient
}
```

\[
w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w, x_i, y_i)
\]
Programming with Spark

• Example: Logistic Regression
  • Logistic regression is an iterative algorithm typically used for classification. Like most classification algorithms, it updates weights iteratively based on the training data.

```
val points = sc.textFile(...).map(parsePoint) // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    g(p) // Apply the function of logistic regression
  }.reduce(_ + _)
  w = alpha * gradient
}
```

Spark starts the execution when the action `reduce` is applied

\[
w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)
\]
Programming with Spark

• Example: Logistic Regression

  • Logistic regression is an iterative algorithm typically used for classification. Like most classification algorithms, it updates weights iteratively based on the training data.

  $\text{points is being re-loaded upon every iteration!} \quad \text{Unnecessary!}$

  \[
  w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)
  \]
Caching and Persistence

• By default, RDDs are recomputed each time you run an action on them. This can be expensive (time-consuming) if you need to use a dataset more than once.

• Spark allows you to control what is cached in memory
  • `persist()` or `cache()`

```scala
val points = sc.textFile(...).map(parsePoint).persist() // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
  val gradient = points.map { p => g(p) // Apply the function of logistic regression }
  .reduce(_+_)
  w <- alpha * gradient
}
```

`points` is loaded once and is cached in memory. It can be re-used on each iteration.
import pyspark

if __name__ == '__main__':
    sc_conf = pyspark.SparkConf() \
        .setAppName('task1') \
        .setMaster('local[*]') \
        .set('spark.driver.memory', '8g') \
        .set('spark.executor.memory', '4g')

    sc = pyspark.SparkContext(conf=sc_conf)
    sc.setLogLevel("OFF")

    input_path = './work-count-sample-doc.txt'
    data = sc.textFile(input_path)
    first10 = data.map(lambda line: line.split(' ')).take(10)

    count = data.flatMap(lambda line: line.split(' ')) \
        .map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b).collect()
Spark SQL

- Spark SQL is a component of Spark Stack
  - A Spark module for structured data processing
  - Implemented as a library on top of Spark

- Advantages
  - Support relational processing in spark
  - High performance
  - Easily support new data sources such as semi-structured data
Spark SQL - DataFrame

• DataFrame is the core abstraction of Spark SQL
  • Conceptually, RDDs are full of records with some known schema
  • DataFrame is like a table in relational database

• Once you have a DataFrame to operate on, you can freely write familiar SQL syntax to operate on your dataset!

```scala
// Register the DataFrame as a SQL temporary view
testDF.createOrReplaceTempView("order")
// This gives the name to the DataFrame in SQL,
// so we can refer it in an SQL FROM statement

val testDF1 = spark.sql("select * from order where date == '2017/01/03'"")
```
Managing Spatial Data in Spark

• Classic - single machine DBMS or GIS tools
  • ArcGIS/QGIS
  • PostgreSQL + PostGIS

• Managing spatial data is not easy in Spark
  • No spatial data type support
  • No spatial index
  • No spatial query
Managing Spatial Data in Spark

• Classic - single machine DBMS or GIS tools
  • ArcGIS/QGIS
  • PostgreSQL + PostGIS

• Managing spatial data is not easy in Spark
  • No spatial data type support
  • No spatial index
  • No spatial query
Apache Sedona (GeoSpark)

- GeoSpark is a cluster computing system for processing large-scale spatial data.
- GeoSpark extends RDDs to Spatial Resilient Distributed Datasets (SRDDs) that efficiently load, process, and analyze large-scale spatial data across machines.
- Spark SQL => Spatial SQL

Spatial RDD (SRDD) Layer

• SRDD supports heterogeneous data sources
  • E.g., CSV, WKT, GeoJSON, NetCDF/HDF, and Shapefile

• SRDD partitioning
  • GeoSpark automatically partitions a loaded Spatial RDD according to its internal spatial data distribution
  • The intuition is to group spatial objects into the same partition based on the spatial proximity, so that reducing the data shuffles across cluster

Yu et al. Spatial data management in apache spark: the GeoSpark perspective and beyond, 2018
Algorithm 1 SRDD spatial partitioning

Data: An original SRDD
Result: A repartitioned SRDD

/* Step 1: Build a global grid file at master node */
1 Take samples from the original SRDD A partitions in parallel;
2 Construct the selected spatial structure on the collected sample at master node;
3 Retrieve the grids from built spatial structures;

/* Step 2: Assign grid ID to each object in parallel */
4 foreach spatial object in SRDD A do
5   foreach grid do
6     if the grid intersects the object then
7        Add (grid ID, object) pair into SRDD B;
8        // Only needed for R-Tree partitioning
9     if no grid intersects the object then
10        Add (overflow grid ID, object) pair into SRDD B;
11 /* Step 3: Repartition SRDD across the cluster */
12 Partition SRDD B by ID and get SRDD C;
13 Cache the new SRDD C in memory and return it;

Yu et al. Spatial data management in apache spark: the GeoSpark perspective and beyond, 2018
Building Local Indexes

• Building a spatial index for the entire dataset is not possible because a tree-like spatial index yields additional 15% storage overhead

• If the user wants to use a spatial index, GeoSpark will build a set of local spatial indexes rather than a single global index
  • Create a spatial index (R-Tree or Quad-Tree) per RDD partition
  • Local indexes can be persisted in memory or disk
Spatial SQL Example

```python
schema_point = StructType() \
    .add("tid", IntegerType(), False) \
    .add("x", DoubleType(), False) \
    .add("y", DoubleType(), False)

def distance_join():
    # 1. self join
    df_all_point = spark.read.option("header", True).schema(schema_point).csv(all_point_file_path)
    df_all_point.createOrReplaceTempView("all_point_import")
    df_all_point1 = spark.sql("SELECT tid, ST_Point(x, y) as point from all_point_import")
    df_all_point1.createOrReplaceTempView("all_point")

    df_join = spark.sql(f""
        SELECT/** BROADCAST(t2) */
            t1.tid AS tid_1,
            t2.tid AS tid_2,
        FROM all_point t1, all_point t2
        WHERE ST_Distance(t1.point, t2.point) < {prec_distance}
            AND t1.tid != t2.tid
        ORDER BY t1.tid, t2.tid;
    ""
    )
    df_join.createOrReplaceTempView("distance_join")
```

Other examples: https://sedona.apache.org/tutorial/sql-python/
Assignment 1

2. Programming Requirements and Environment Settings
   a. You must use SQL and Python to implement all tasks.
   b. Programming Environment:
      o JAVA version 1.8, Python 3.7, Pyspark 3.0.0, Sedona 1.1.1
      o [Optional] You can use Conda to manage your programming environment.
        $conda create --name [ENV] -y python=3.7
        $conda activate [ENV]
        $conda install -c conda-forge gdal==3.4.0
        $conda install -c conda-forge pyspark==3.0.0
        $pip install apache-sedona

Sedona Python requires two additional jar packages, sedona-python-adaptor and geotools-wraper, to work properly. Specifically, you need to put two jar packages under [YOUR PYTHON PATH]/site-packages/pyspark/jar/.