**1. Monte Carlo simulation:** a method of estimating the mathematical constant Pi (π) using a Monte Carlo simulation,

Draw a square with a circle inscribed inside it.

Randomly generate a large number of points within the square.

Count how many of these points fall inside the circle.

Divide the number of points inside the circle by the total number of points, then multiply by 4 to get an approximation of Pi.

**2. Distributed File System**

Large file is divided into chunks

Each chunk is replicated into different servers

Master node maintains the (file, chunk server) lookup table

**3.Local Ubuntu**:

Creating an Ingress Rule:

Open the navigation menu and click Networking, and then click Virtual Cloud Networks.

• Select the VCN you created with your compute instance.

• With your new VCN displayed, click <your-subnet-name> subnet link. The public subnet information is displayed with the Security Lists at the bottom of the page. A link to the Default Security List for your VCN is displayed.

• Click the Default Security List link. The default Ingress Rules for your VCN are displayed.

• Click Add Ingress Rules. An Add Ingress Rules dialog is displayed.

• Fill in the ingress rule form and Click Add Ingress Rules.

131.96.0.0/16 is the public IP range of GSU

• 10.0.0.0/16 is the private IP range of the VCN

• Needs port 22 for SSH

• Needs to open ports 4040, 8080, 8081 for Spark web GUI

• For simplicity in setting, we choose to open all ports

**6. MapReduce: programming model**

Map

 Accepts input key/value pair

 Emits intermediate key/value pair

Reduce  
¬ Accepts intermediate key/value\* pair  
¬ Emits output key/value pair

GFS (Google File System)

Word Counting:

map(string value)

// key: document name

// value: document contents

for each word w in value

EmitIntermediate(w, “1”);

reduce(string key, iterator values)

// key: word

// values: list of counts

int results = 0;

for each v in values

result += ParseInt(v);

Emit(AsString(result));

Map

• Process a key/value pair to generate intermediate key/value pairs

Reduce

• Merge all intermediate values associated with the same key

Partition

• By default : hash(key) mod R

• Well balanced

**Hadoop:** Java-based implementation of MapReduce. Use HDFS (Hadoop Distributed File System) as underlying file system

Divide and Conquer: Recursion

• Note easy to implement in MapReduce / Distributed Computing

• Some steps can be computed by using MapReduce

• Several MapReduce processes can be chained

**7. Spark: Transformations and actions on data sets. Data flow, or lineage graph among data sets, induced by the transformations**

Resilient Distributed Datasets: data sets in Spark

• Resilient Distributed Dataset

 One RDD has one or more (logical) partition

 Partitions are Distributed/Computed across machines

 Rebuilt from base data on failure (versus replication)

 Lazy evaluation - created on demand

• RDD types offer various functions (Any type of Python, Java, or Scala objects, including user-defined classes)

 map, reduce

 groupBy, reduceByKey

 joins (inner, leftOuterJoin, rightOuterJoin)

 filter, sample

Components of Spark: (Spark SQL, Spark Streaming, MLlib, Graphx)

Apache Spark Core

 Underlying general execution engine for Spark platform that all other functionality is built upon

 Provides In-Memory computing and referencing datasets in external storage systems

• Spark SQL

 Introduces a new data abstraction called SchemaRDD

 Provides support for structured and semi-structured data

• Spark Streaming

 Leverages Spark Core’s fast scheduling capability to perform streaming analytics

 Ingests data in mini-batches and performs RDD transformation on those mini-batches of data

MLlib (Machine Learning Library)

 Distributed machine learning framework above Spark because of the distributed memory-based Spark architecture

 Spark MLlib is nine times as fast as the Hadoop disk-based version of Apache Mahout

 Apache Mahout: Scalable machine learning and data mining

GraphX

 Distributed graph-processing framework on top of Spark

 Provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API

 Also provides an optimized runtime for this abstraction

In the current MapReduce framework, the only way to reuse data between computations (two MapReduce jobs) is to Write it to an external stable storage system (e.g., HDFS)

• Most of the Hadoop applications spend more than 90% of the time doing HDFS read-write operations.

• Iterative and Interactive applications require faster data sharing across parallel jobs. In MapReduce, it is slow due to replication, serialization, and disk IO.

word = rdd.filter(lambda s: "error" in s)

def containsError(s):

return "error" in s

word = rdd.filter(containsError)

**8. Ray Programming Model**

Task: Task is the special function of Ray version. The execution of the remote function is called Task. The remote function in Ray is as stateless as a traditional function, which just processes the input and returns the result without intermediate state.

Actor: Actor is the special class of Ray version. When an Actor is initialized, Ray will see it as a separate process and execute it remotely. All methods in an Actor will be treated as Task. However, the most

different part between these methods in an Actor and Task is that they execute on a stateful Worker, which includes global states and can be modified by various tasks within Ray.

Python Function => Ray Tasks

import ray

import numpy as np

ray.init()

@ray.remote

def make\_array(...):

a = ... # Construct a NumPy array

return a

@ray.remote

def add\_arrays(a, b):

return np.add(a, b)

ref1 = make\_array.remote(...)

ref2 = make\_array.remote(...)

ref3 = add\_arrays.remote(ref1, ref2)

ray.get(ref3)

Python Class => Ray Actor

@ray.remote  
class Counter(object):  
 def \_ \_init\_ \_(self):  
 self.value = 0  
 def increment(self):  
 self.value += 1  
 return self.value  
 def get\_count(self):  
 return self.value

MyCounter = Counter.remote()

ref4 = MyCounter.increment.remote()

ref5 = MyCounter.increment.remote()

ray.get([ref4, ref5]) # [1, 2]

Ray is the new state-of-the-art for distributed computing

• The shortest path from your laptop to the cloud

• Run complex distributed tasks on large clusters from simple code on your laptop

**9. pyspark**

>>> lines = sc.textFile(“/home/rob/Assignment1/test.txt") # Create an RDD called lines

>>> lines.count() # Count the number of items in this RDD

>>> lines.first() # First item in this RDD, i.e. first line of README.md

SparkContext Object

• Driver programs access Spark through a “SparkContext” object

• a “SparkConext” is automatically created for you as the variable called sc

Python – Initializing Spark

from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("My App") # or

conf = SparkConf().setMaster("spark://10.0.0.177:7077").setAppName("My App")

sc = SparkContext(conf = conf)

Initializing a SparkContext

• Minimal way to initialize a SparkContext

• We pass two parameters:

 “local”: a cluster URL. It tells Spark how to connect to a cluster. It runs Spark on one thread on the local machine, without connecting to a cluster

 “spark://10.0.0.177:7077”: a cluster head node URL. This will allow the program run Spark on the multi-node cluster.

 “My App”: An application name. This will identify your application on the cluster manager’s UI if you connect to a cluster.

After the initialization of a SparkContext, you can use all the methods we showed before to create RDDs and manipulate them

• To shutdown Spark,

 call Stop() method on your SparkContext

 Simply exit the application (e.g., with System.exit(0) or sys.exit())

Passing Functions to the Spark Operators:

>>> lines = sc.textFile("README.md")

>>> pythonLines = lines.filter(lambda line: "Python" in line)

>>> pythonLines.first()

def hasPython(line):

return “Python” in line

>>> pythonLines = lines.filter(hasPython)

>>> pythonLines.first()

Function-based Operations like “filter” also parallelize across the cluster.

• Spark automatically takes your function (e.g., line.contains("Python")) and ships it to executor nodes.

• You can write code in a single driver program and automatically have parts of it run on multiple nodes.

Components for Distributed Execution:

Driver programs manage a number of nodes called executors

• For example, “lines.count()”

• Different machines will count lines in different ranges of the file

• If we run the Spark shell locally, it executed all its work on a single machine

**10. Spark Runtime Architecture:**

The Driver

• The main() method runs in this Java process (driver).

• It creates a SparkContext, creates RDDs, and performs transformations and actions.

• When you launch a Spark shell, you have created a driver program

Two Duties

• Converting a user program into tasks

• Scheduling tasks on executors

pyspark

book = sc.textFile(“/home/rob/data/peterpan.txt”)

book.first()

book.count()

Executors

• Spark executors are worker processes responsible for running the individual tasks in a given Spark job.

Two Roles:

• They run the tasks and return results to the driver

• They provide in-memory storage for RDDs that are cached by user programs

Launching a Program – spark-submit

1. The user submits an application using spark-submit.

2. spark-submit launches the driver program and invokes the main() method specified by the user.

3. The driver program contacts the cluster manager to ask for resources to launch executors.

4. The cluster manager launches executors on behalf of the driver program.

5. The driver process runs through the user application. Based on the RDD actions and transformations in the program, the driver sends work to executors in the form of tasks.

6. Tasks are run on executor processes to compute and save results.

7. If the driver’s main() method exits or it calls SparkContext.stop(), it will terminate the executors and release resources from the cluster manager.

Submitting a Python application: spark-submit my\_script.py

# submit the python file to a local spark cluster

spark-submit ~/code/spark\_pi.py

# submit the python file to a spark cluster at spark://192.168.56.3:7077

spark-submit --master spark://192.168.56.3:7077 ~/code/spark\_pi.py

# change the number of executors and number of CPU cores

spark-submit --master spark://192.168.56.3:7077 --num-executors 8 --executor-cores 1 --executor-memory 512M ~/code/spark\_pi.py

# Submitting a Java application to Standalone cluster mode

$ ./bin/spark-submit \

--master spark://hostname:7077 \

--deploy-mode cluster \

--class com.databricks.examples.SparkExample \

--name "Example Program" \

--jars dep1.jar,dep2.jar,dep3.jar \

--total-executor-cores 300 \

--executor-memory 10g \

myApp.jar "options" "to your application" "go here"

# Submitting a Python application in YARN client mode

$ export HADOOP\_CONF\_DIR=/opt/hadoop/conf

$ ./bin/spark-submit \

--master yarn \

--py-files somelib-1.2.egg,otherlib-4.4.zip,other-file.py \

--deploy-mode client \

--name "Example Program" \

--queue exampleQueue \

--num-executors 40 \

--executor-memory 10g \

my\_script.py "options" "to your application" "go here"

**11. Programming with RDDs**

RDD (Resilient Distributed Dataset)

• Spark’s core abstraction for working with data

• It is simply a distributed collection of elements

• Creating new RDDs

• Transforming existing RDDs

• Calling operations on RDDs

RDD Features

• Each RDD is split into multiple partitions

• Each partition may be computed on different computer nodes

• RDD can contain any type of Python, Java, or Scala objects, including user-defined classes

• Loading an external dataset

• Distributing a collection of objects

Basic Usage

• Creating an RDD with textFile() in Python

• book = sc.textFile(“/home/rob/data/peterpan.txt”)

• book = sc.textFile(“hdfs://localhost:9000/user/rob/data/peterpan.txt”)

• Calling the filter() transformation

• peterLines = book.filter(lambda line: “Peter” in line)

• Calling the first() action

• peterLines.first()

Spark’s RDD

• Spark computes RDDs only in a lazy fashion

• That is, The first time they are used in an action.

• Spark sees the whole chain of transformations, and computes just the data needed for the result.

• By default, Spark’s RDDs are recomputed each time you run an action on them.

• If you want to reuse an RDD in multiple actions, you can ask Spark to persist it using RDD.persist()

• Logic: If you will not reuse the RDD, there is no reason to waste storage space. Spark could instead stream through the data once!

Persisting an RDD in memory

• book = sc.textFile(“/home/rob/data/peterpan.txt”)

• peterLines = book.filter(lambda line: “Peter” in line)

• peterLines.persist()

• peterLines.count()

• peterLines.first()

General Workflow of Spark Programs

1. Create some input RDDs from external data.

2. Transform them to define new RDDs using transformations like filter().

3. Ask Spark to persist() any intermediate RDDs that will need to be reused.

4. Launch actions such as count() and first() to kick off a parallel computation, which is then optimized and executed by Spark.

Creating RDDs

• Two ways:

1. Loading an external dataset

2. Parallelizing a collection in the driver program

python: lines = sc.parallelize(["pandas", "i like pandas"])

scala: val lines = sc.parallelize(List("pandas", "i like pandas"))

java: JavaRDD<String> lines = sc.parallelize(Arrays.asList("pandas", "i like pandas"));

RDD Operations

• Two types of operations:

1. Transformations : map() and filter()

2. Actions : count() and first()

inputRDD = sc.textFile("log.txt")

errorsRDD = inputRDD.filter(lambda x: "error" in x)

warningsRDD = inputRDD.filter(lambda x: "warning" in x)

badLinesRDD = errorsRDD.union(warningsRDD)

action: Each time we call a new action, the entire RDD must be computed “from scratch”. We can persist an RDD.

inputRDD = sc.textFile("log.txt")

errorsRDD = inputRDD.filter(lambda x: "error" in x)

warningsRDD = inputRDD.filter(lambda x: "warning" in x)

badLinesRDD = errorsRDD.union(warningsRDD)

print "Input had " + badLinesRDD.count() + " concerning lines"

print "Here are 10 examples:"

for line in badLinesRDD.take(10):

print line

badLinesRDD.collect() #The entire dataset must fit in memory on a single machine of the  
driver program. Collect() should not be used on large dataset.

Saving the data

badLinesRDD.collect() #Collect (Action) - Return all the elements of the dataset as an array at the driver program. The entire dataset must fit in memory on a single machine. Collect() should not be used on large dataset.

badLinesRDD.saveAsTextFile()

badLinesRDD.saveAsSequenceFile() #Write data out to a distributed storage system such as HDFS or Amazon S3

Passing Functions to Spark in Python

word = rdd.filter(lambda s: "error" in s)

def containsError(s):  
 return "error" in s  
word = rdd.filter(containsError)

Common Transformations

Two most common transformations:

1. map()

Input type can be different from

the output type

 Example Input: String

 Example Output: Double

nums = sc.parallelize([1, 2, 3, 4])  
squared = nums.map(lambda x: x \* x).collect()  
for num in squared:  
print "%i " % (num)

2. filter()

Actions

• The most common action: reduce()

sum = rdd.reduce(lambda x, y: x + y)

• fold() similar to reduce() but has a initial value for the initial call on each partition

• Initial value: 0 for + ; 1 for \* ; or {} for concatenation

sum = rdd.fold(0)(lambda x, y: x + y)

product = rdd.fold(1)(lambda x, y: x \* y)

• For reduce() and fold()

• Input type can be the same as the output type

 Example Input: Integer

 Example Output: Integer

aggregate()

• Suppose we want to compute the average

sumCount = nums.aggregate((0, 0),

(lambda acc, value: (acc[0] + value, acc[1] + 1)),

(lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])))

return sumCount[0] / float(sumCount[1])

• output type can be different from the Input type

Persistence (Caching) - Motivation

• Spark RDDs are lazily evaluated (lazy evaluation)

• We need to use the same RDD multiple times

• Expensive for iterative algorithms, which look at the data many times

• The nodes that compute the RDD store their partitions.  
• If one node fails, Spark will re-compute the lost partitions of data.  
• We can also replicate the data on multiple nodes for faster speed.  
Spark has many (5) levels of persistence to choose from  
Default persist(): store the data in the JVM heap as unserialized objects.

val result = input.map(x => x \* x)

println(result.count())

println(result.collect().mkString(","))

• Serialization refers to converting objects into a stream of bytes and vice-versa (de-serialization) in an optimal way to transfer it over nodes of network or store it in a file/memory buffer.

• Spark provides two serialization libraries and modes are supported and configured through spark.serializer property.

Persist() and Cache()

• With cache(), you use only the default storage level MEMORY\_ONLY.

• With persist(), you can specify which storage level you want.

• Cache() is a synonym of persist() or persist(StorageLevel.MEMORY\_ONLY), i.e. cache is merely persist with the default storage level MEMORY\_ONLY.

persist() in Scala

import org.apache.spark.storage.StorageLevel

val result = input.map(x => x \* x)

result.persist(StorageLevel.DISK\_ONLY)

println(result.count())

println(result.collect().mkString(","))

• We call persist() on the RDD before the first action.  
• The persist() call on its own does not force evaluation.  
• unpersist() lets you manually remove an RDD from the cache.