Spark and Scalable Machine Learning

Guru Medasani

Class structure

- ❖ 30 minutes - Spark Motivation and Intro
- ❖ 5 minutes - Break
- ❖ 20 minutes - PySpark Demo
- ❖ 45 minutes - ML Motivation and Intro
- ❖ 30 minutes - ML Demo
- ❖ 10 minutes - Questions

About me

Solutions Architect - Cloudera

Past:
BigData Engineer - Monsanto
Hadoop Admin - Monsanto
Linux Admin - Monsanto


UTV

What's in this talk?

- Motivation for Spark
- Introduction to Apache Spark
- Motivation for Machine Learning
- Introduction to Machine Learning
- Basic Linear Algebra Review
- Build a Linear Regression algorithm from scratch in Spark

What is BigData?

big data *n. Computing* (also with capital initials) data of a very large size, typically to the extent that its manipulation and management present significant logistical challenges; (also) the branch of computing involving such data.

[Oxford-dictionary-big data]

Apache Spark

Motivation

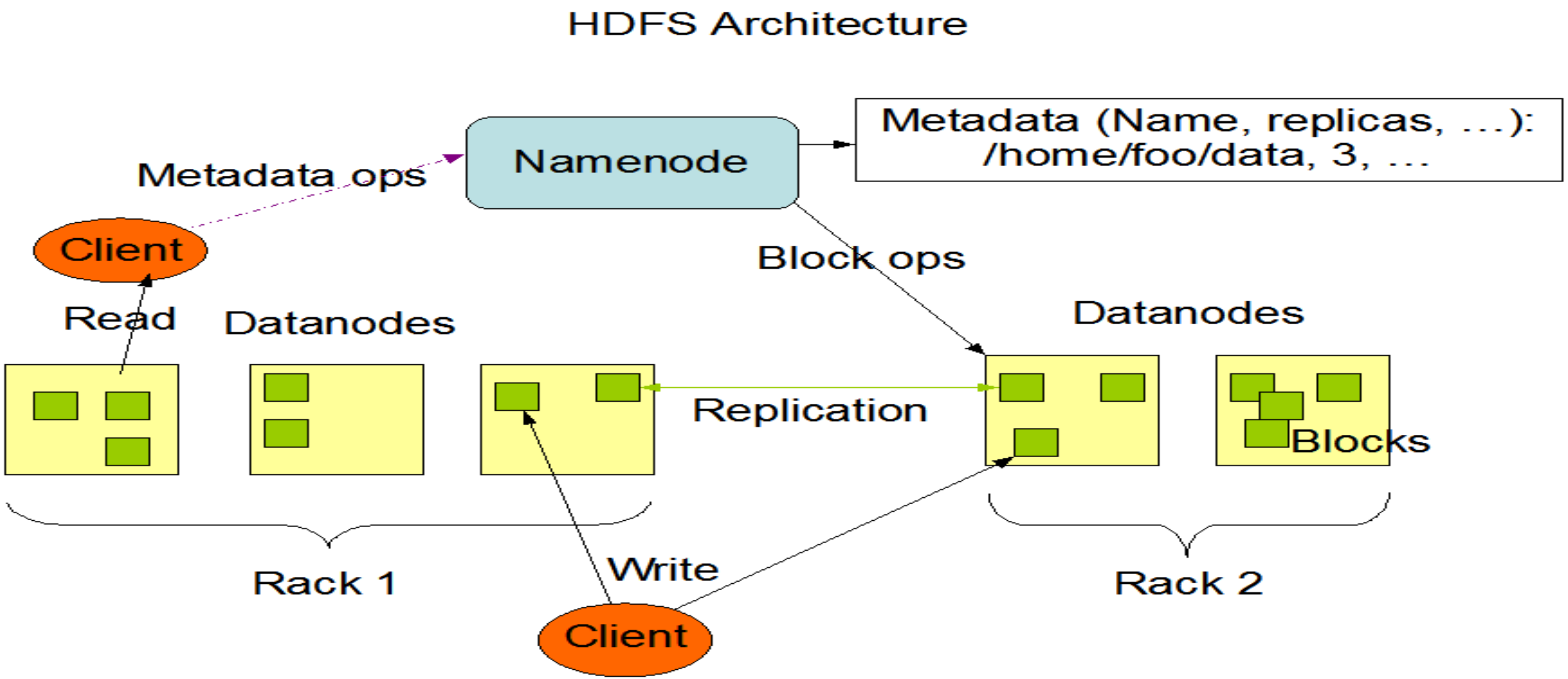
Hadoop

- Created by Doug Cutting and Mike Cafarella in 2005
- Distributed storage and processing system
- Based on two Google Papers
  - The Google File System
  - MapReduce: Simplified Data processing on Large Cluster

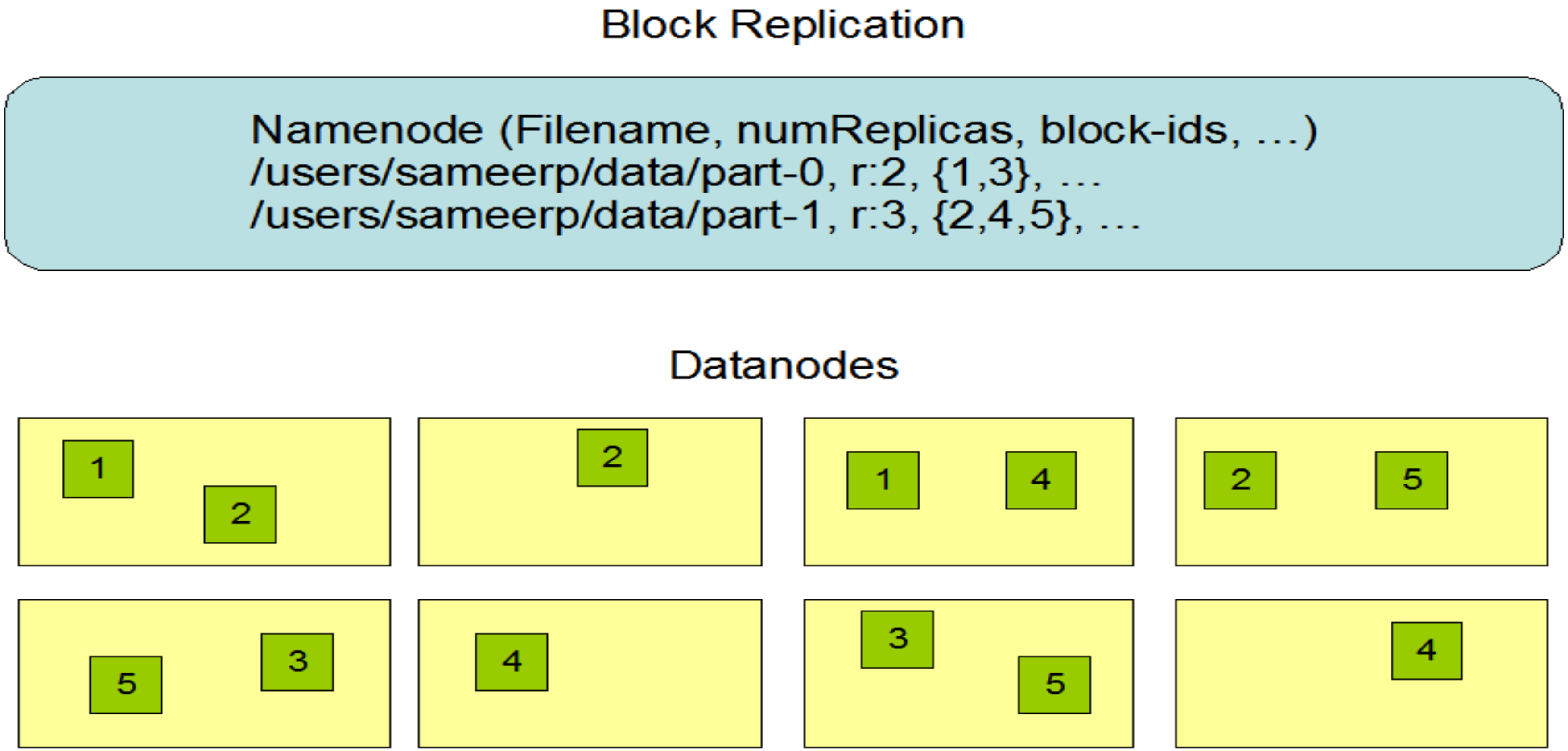
Hadoop - components

- **HDFS** - Hadoop Distributed File System
  - tuned for large datasets
  - highly fault-tolerant
  - highly scalable
- **MapReduce**
  - software framework for parallel processing
  - highly fault-tolerant
  - highly scalable

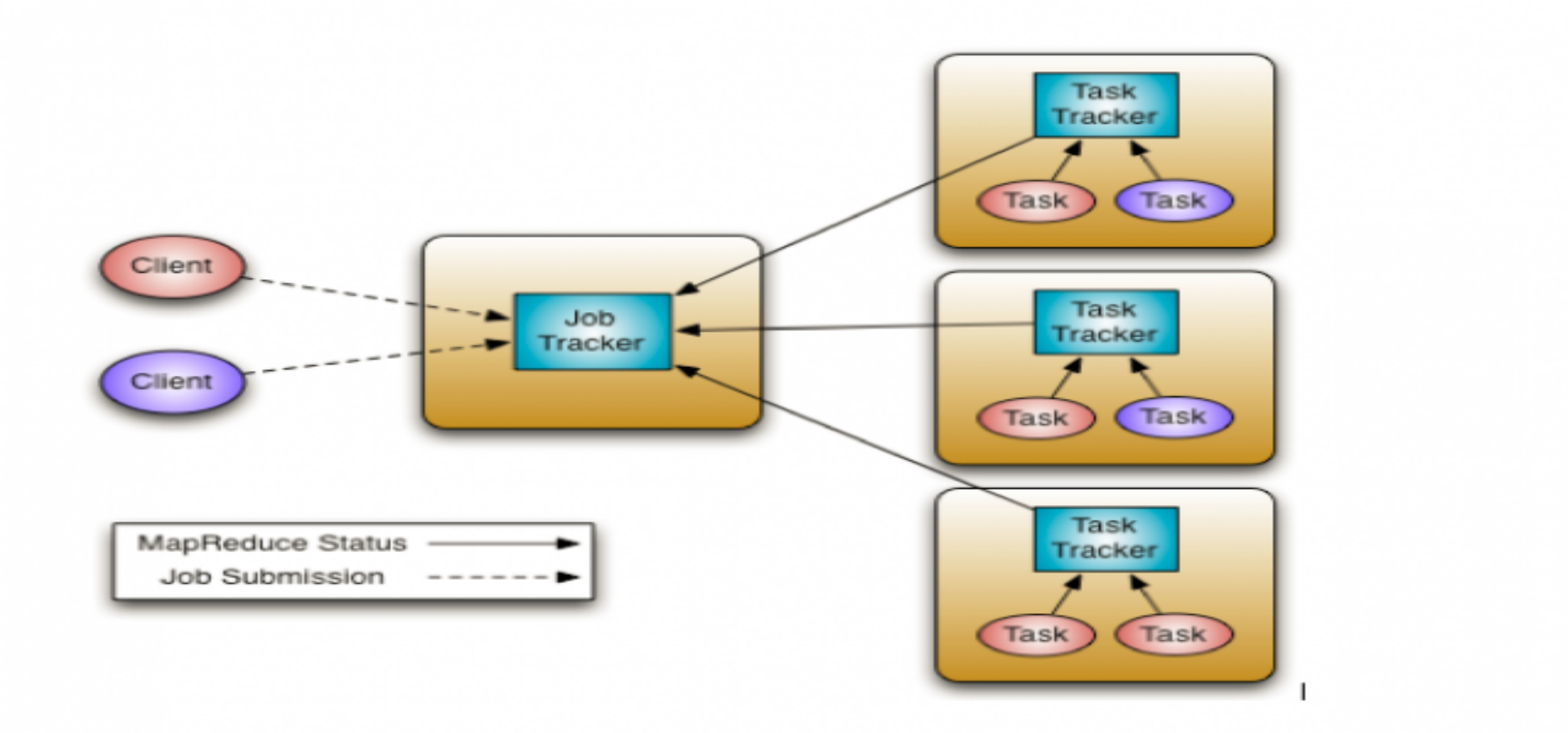
HDFS - high level



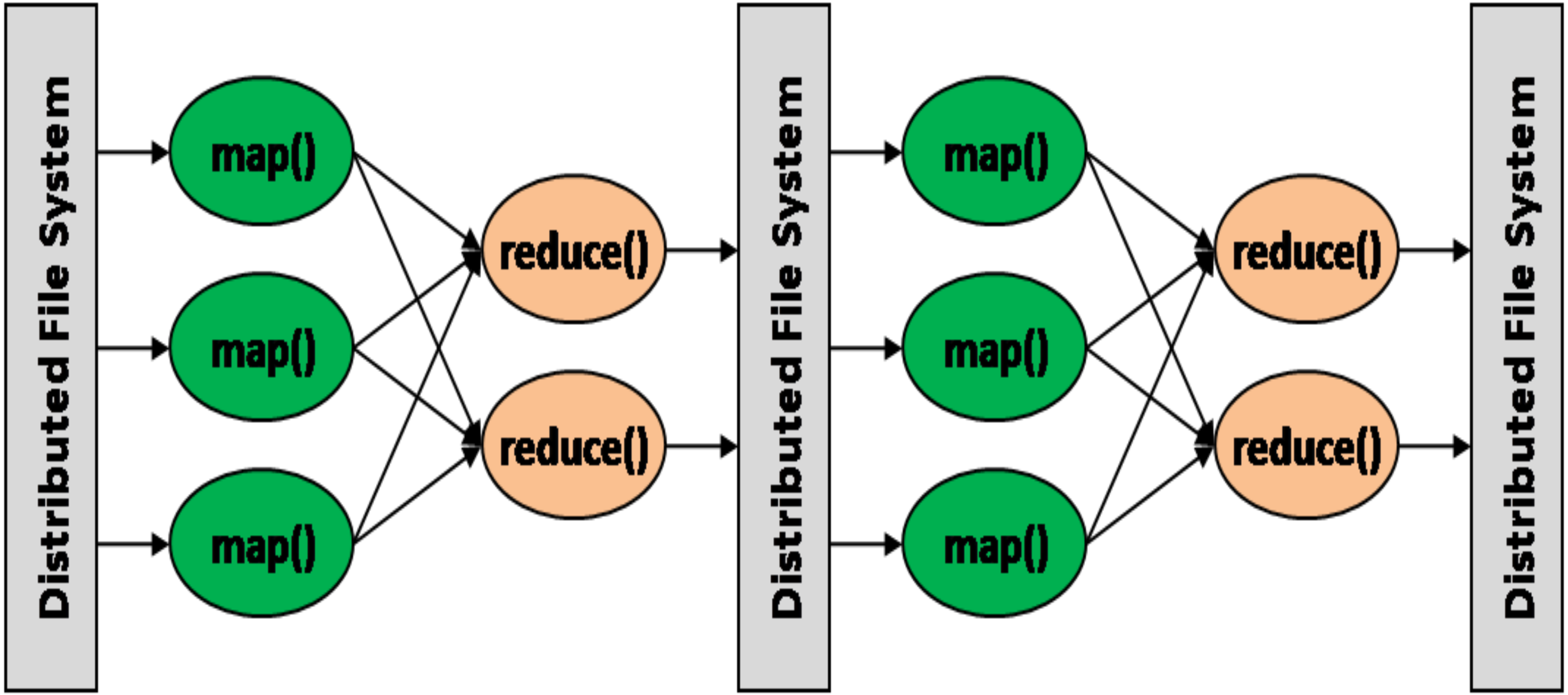
HDFS - data replication



MapReduce - high level



MapReduce – Distributed Execution



Drawbacks of MapReduce

- Designed for batch-oriented jobs/workflows
- Reads/writes data from disk too many times
- Hard to do near real-time processing
- Can’t store results in-memory
- Hard to do iterative algorithms
- Disk I/O is very slow

Opportunity

- Keep more data in-memory

- Create new distributed execution engine

Apache Spark

Introduction

Apache Spark - what is it?

- open-source cluster computing software
- started in 2009
- originally developed in AMPLab - UC Berkeley
- runs on hadoop
- apache top-level project since February 2014
- supported by major hadoop distributions like Cloudera
- can scale to 1000’s of nodes

Spark - what does it provide?

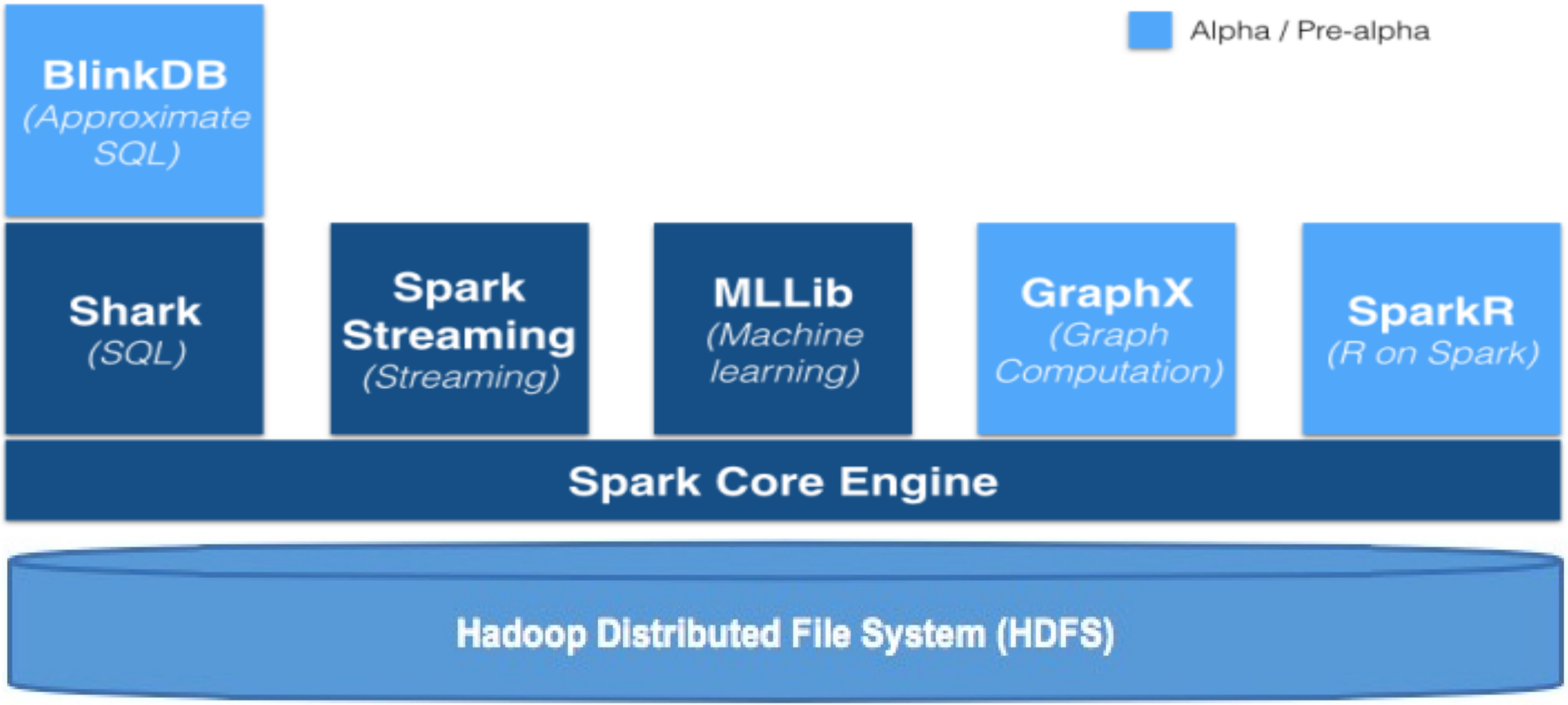
- fast and general-purpose processing engine
- runs computations in memory
- significantly faster than mapreduce
- provides more operators in addition to map and reduce
- flatMap, join, reduceByKey, count, groupByKey, sortByKey
- new operators added very frequently
- provides simple APIs in Java, Scala and Python
- provides rich higher-level components

Spark - how fast can it process?

- 2014 Daytona GraySort Winner
- cluster size - 206 EC2 machines
- Sorted 100TB of data on disk in 23 minutes
- generates 500 TB of disk I/O
- generates 200TB of network I/O
- 3x faster than mapreduce
- 10x fewer machines than mapreduce

	Hadoop MR Record	**Spark Record**	**Spark 1PB**
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

Spark - A Unified Stack



Spark Core - Intro

- provides basic functionality of Spark
  - SparkContext - main entry point to Spark
- contains main programming abstraction
  - RDD - Resilient Distributed Datasets
- task scheduling
- memory management
- access to storage systems
- fault recovery

Spark-Shell - Intro

- REPL (Read Eval Print Loop) for Spark
- built on top of Scala shell
- data exploration
- interactive analysis of data
- prototyping spark applications

Using spark-shell

Scala

```
$ ./bin/spark-shell --master local[4]
```

```
$ ./bin/spark-shell --master local[4] --jars code.jar
```

Python

```
$ ./bin/pyspark --master local[4]
```

```
$ ./bin/pyspark --master local[4] --py-files code.py
```

SparkContext

- main entry point for Spark functionality
- represents the connection to a Spark cluster
- create RDDs
- created in driver program
- automatically created in spark-shell

Initializing Spark - Scala

```scala
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
val conf = new SparkConf().setAppName(appName).setMaster(master)
val sc = new SparkContext(conf)
```

Initializing Spark - Java

```java
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;
SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);
JavaSparkContext sc = new JavaSparkContext(conf);
```

Initializing Spark - Python

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

RDD - Resilient Distributed Datasets

- basic programming abstraction
- immutable, partitioned collection of elements
- can be operated in parallel
- five properties:
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs
  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)

Create RDDs

Two ways to create RDDs

- parallelize an existing collection
- load a dataset from external storage system
    - HDFS
    - HBase
    - Cassandra
    - S3
    - local filesystem

Parallelize Collections

Scala

```scala
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

Java

```java
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

Python

```python
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

External Datasets

- create RDDs from different file formats
  - Textfiles
  - Sequence Files
  - Hadoop InputFormat
  - Avro (open-source from Cloudera)
  - Parquet (open-source from Cloudera & Twitter)

External Datasets - contd

Scala

```
scala> val distFile = sc.textFile("data.txt")
distFile: RDD[String] = MappedRDD@1d4cee08
```

Java

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

Python

```
>>> distFile = sc.textFile("data.txt")
```

RDD Operations

- RDDs support two types of Operations
  - Transformations
    - create a new dataset from existing one
    - Ex: map(), filter()
  - Actions
    - perform a computation and return a result
    - Ex: count(), first()

RDD Operations - contd

- transformations are lazy
- transformations are remembered as a dependency list
- only computed when an action is applied
- but recomputed each time an action is performed
    - means dataset is read from disk every time
- for iterative algorithms, we can persist the mapped data
    - store the dataset in memory/disk
    - can be replicated to multiple nodes

RDD Operations - contd

```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)
scala> val distData = sc.parallelize(data)
distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:14
scala> val newData = distData.map(x => x*2)
newData: org.apache.spark.rdd.RDD[Int] = MappedRDD[1] at map at <console>:16
scala> val newData1 = newData.map(x => x*100)
newData1: org.apache.spark.rdd.RDD[Int] = MappedRDD[2] at map at <console>:18
scala> newData1.toDebugString
res3: String =
(4) MappedRDD[2] at map at <console>:18
 |  MappedRDD[1] at map at <console>:16
 |  ParallelCollectionRDD[0] at parallelize at <console>:14
```

Spark Web UI

Spark | Stages | Storage | Environment | Executors | **Spark shell** application UI

Spark Stages

Total Duration: 31 min
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 9
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write

Completed Stages (9)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
8	collect at <console>:21 +details	2014/11/17 19:32:21	13 ms	4/4	216.0 B		
7	toArray at <console>:21 +details	2014/11/17 19:31:43	16 ms	4/4	216.0 B		
6	foreach at <console>:21 +details	2014/11/17 19:31:07	15 ms	4/4	216.0 B		
5	foreach at <console>:21 +details	2014/11/17 19:31:02	22 ms	4/4	216.0 B		

Spark | Stages | Storage | Environment | Executors | **Spark shell** application UI

Details for Stage 8

Total task time across all tasks: 8 ms
Input: 216.0 B

Summary Metrics for 4 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	1 ms	1 ms	1 ms
Duration	1 ms	2 ms	2 ms	3 ms	3 ms
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	0 ms
Scheduler delay	8 ms	8 ms	9 ms	10 ms	10 ms
Input	48.0 B	48.0 B	48.0 B	72.0 B	72.0 B

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input	Shuffle Read	Shuffle Write	Shuffle Spill (Memory)	Shuffle Spill (Disk)
localhost	CANNOT FIND ADDRESS	43 ms	4	0	4	216.0 B	0.0 B	0.0 B	0.0 B	0.0 B

Tasks

Index	ID	Attempt	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Accumulators	Input	Errors
0	32	0	SUCCESS	ANY	localhost	2014/11/17 19:32:21	1 ms			48.0 B (memory)	
2	34	0	SUCCESS	ANY	localhost	2014/11/17 19:32:21	2 ms			48.0 B (memory)	
1	33	0	SUCCESS	ANY	localhost	2014/11/17 19:32:21	2 ms			48.0 B (memory)	
3	35	0	SUCCESS	ANY	localhost	2014/11/17 19:32:21	3 ms			72.0 B (memory)	

Transformations

- map()
- filter()
- flatMap()
- mapPartitions()
- mapPartitionsWithIndex()
- sample()
- union()
- intersection()
- distinct()
- groupByKey()
- reduceByKey()
- aggregateByKey()
- sortByKey()
- join()
- cogroup()
- cartesian()
- pipe()
- coalesce()
- repartition()

Actions

- reduce()
- collect()
- count()
- first()
- take()
- takeSample()
- takeOrdered()
- takeSample()
- takeOrdered()
- saveAsTextFile()
- saveAsSequenceFile()
- saveAsObjectFile()
- countByKey()
- foreach()

RDD Persistence

- persist large datasets in memory
- often 10x faster during actions
- important for interactive and iterative algorithms
- caching is fault-tolerant
- can persist RDD using persist() or cache()
- cache is LRU least-recently-used fashion
- unpersist() to manually remove RDD from cache

RDD persist() Storage Levels

- MEMORY_ONLY
- MEMORY_AND_DISK
- MEMORY_ONLY_SER
- MEMORY_AND_DISK_SER
- DISK_ONLY
- MEMORY_ONLY_2
- MEMORY_AND_DISK_2
- OFF_HEAP(experimental)

Spark Streaming - Intro

- extension of Spark Core API
- scalable, high-throughput, fault-tolerant stream processing
- supports several data sources
  - Kafka
  - Flume
  - Twitter
  - ZeroMQ
  - Kinesis
  - TCP sockets

Spark Streaming - contd

Break

5 minutes

Apache Spark

Demo

Machine Learning

Motivation

Questions?

Thank you

email: gdmeda@outlook.com
Twitter: @gurumedasani