PART 1. BATCH COMPUTING MODEL FOR BIG DATA ANALYTICS

3. IN-MEMORY CLUSTER COMPUTING

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

FAQs

- Programming Assignment 1
  - We had discuss link analysis in week 3
  - We will discuss Apache Spark in this week
  - Installation/configuration guidelines for Hadoop and Spark has been uploaded
  - Port assignment has been posted
  - Use Scala or Java

- Saptashwa Mitra will teach on Thursday

Today’s topics

- Link analysis
  - Introduction to Apache Spark
  - RDD

3. In-Memory Cluster Computing: Apache Spark

This material is built based on

- Spark programming guide
  - https://spark.apache.org/docs/2.2.0/
  - Job Scheduling
    - https://spark.apache.org/docs/2.0.0-preview/job-scheduling.html
Distributed processing with the Spark framework

- Spark contains multiple closely integrated components
  - Spark core
    - Computational engine
    - Scheduling, distributing, and monitoring applications
  - Spark Streaming
    - Processes live streams of data
- MLlib
  - Machine learning functionality
  - ML algorithms (classification, regression, clustering and collaborative filtering)
  - Model evaluation
  - Data import

Inefficiencies for emerging applications:

1. Data reuse
   - Data reuse is common in many iterative machine learning and graph algorithms
   - e.g. PageRank, K-means clustering, and logistic regression

2. Interactive data analytics
   - User runs multiple ad-hoc queries on the same subset of the data

Existing approaches

- Hadoop
  - Writing output to an external stable storage system
  - e.g. HDFS
  - Substantial overheads due to data replication, disk I/O, and serialization
- Pregel
  - Iterative graph computations
- HaLoop
  - Iterative MapReduce interface
- Pregel/HaLoop support specific computation patterns
  - e.g. looping a series of MapReduce steps

A unified stack

- GraphX
  - Library for manipulating graphs
  - Performs graph-parallel computations
  - Extends the Spark RDD API
- Cluster Managers
  - Spark can run over a variety of cluster managers
  - Hadoop YARN, Apache Mesos, and Spark built-in cluster manager (Standalone scheduler)
Running a simple example

```scala
/* SimpleApp.scala */
import org.apache.spark.sql.SparkSession
object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system
    val spark = SparkSession.builder.appName("Simple Application").getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    spark.stop()
  }
}
```

**Sbt build file**

```scala
name := "Simple Project" version := "0.0.1" scalaVersion := "2.11.8"
// additional libraries
libraryDependencies ++ = Seq(
  "org.apache.spark" %% "spark-sql" % "2.2.0"
)
```

Scala build and run

```
# Your directory layout should look like this
$ find .
./build.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala

# Package a jar containing your application
$ sbt package
...
[info] Packaging ../[..]/target/scala-2.11/simple-project_2.11-1.0.jar

# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \\n--class "SimpleApp" \\n--master local[4] \\ntarget/scala-2.11/simple-project_2.11-1.0.jar
Lines with a: 46, Lines with b: 23
```

3. In-Memory Cluster Computing: Apache Spark

**RDD (Resilient Distributed Dataset)**

- Read-only, memory resident partitioned collection of records
  - A fault-tolerant collection of elements that can be operated on in parallel
  - RDDs are the core unit of data in Spark
  - Most Spark programming involves performing operations on RDDs

**Overview of RDD**

- **Lineage**
  - How it was derived from other dataset to compute its partitions from data in stable storage?
  - RDDs do not need to be materialized at all times

- **Persistence**
  - Users can indicate which RDDs they will reuse and the storage strategy

- **Partitioning**
  - Users can specify the partitioning method across machines based on a key in each record
Creating RDDs

- Loading an external dataset
  ```scala
  val lines = sc.textFile("/path/to/README.md")
  ```

- Parallelizing a collection in your driver program
  ```scala
  val lines = sc.parallelize(List("pandas", "I like pandas"))
  ```

9/12/2017

Spark Programming Interface to RDD

- Transformations
  - Operations that create RDDs
  - e.g. map, filter, and join
  - RDDs can only be created through deterministic operations on other RDDs or data in stable storage

- Actions
  - Operations that return a value to the application or export data to a storage system
  - e.g. count: returns the number of elements in the dataset
  - e.g. collect: returns the elements themselves
  - e.g. save: outputs the dataset to a storage system

Spark Programming Interface to RDD

- Persist
  - Indicates which RDDs they want to reuse in future operations
  - Spark keeps persistent RDDs in memory by default
  - If there is not enough RAM
    - It can spill them to disk
  - Users are allowed to:
    - Store the RDD only on disk
    - Replicate the RDD across machines
    - Specify a persistence priority on each RDD

Persist

```scala
1: val lines = sc.textFile("data.txt")
2: val lineLengths = lines.map(s => s.length)
3: val totalLength = lineLengths.reduce((a, b) => a + b)
```
Lazy Evaluation
- Transformations on RDDs are lazily evaluated
- Spark will not begin to execute until it sees an action
- Spark internally records metadata to indicate that this operation has been requested
- Loading data from files into an RDD is lazily evaluated
- Reduces the number of passes it has to take over our data by grouping operations together

Passing Functions to Spark
- You can define object MyFunctions and then pass MyFunctions.func1, as follows:

```scala
object MyFunctions {
  def func1(s: String): String = {...}
}
myRdd.map(MyFunctions.func1)
```

- Passing a reference to a method in a class instance (as opposed to a singleton object) is allowed
- This requires sending the object that contains that class along with the method

```scala
class MyClass {
  val field = "Hello"
  def doStuff(rdd: RDD[String]): RDD[String] = {
    rdd.map(x => field + x)
  }
}
```

Transformations
- Transformed RDDs are computed lazily, only when you use them in an action
- Example
  - Suppose that we have a log file log.txt
  - A number of messages, and we want to select only the error messages

```scala
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
```

- Example-continued
  - Print out the number of lines that contained either error or warning
  - union() operates on two RDDs instead of one
  - Transformations can operate on any number of input RDDs

```scala
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
val warningsRDD = inputRDD.filter(line => line.contains("warning"))
val badLinesRDD = errorsRDD.union(warningsRDD)
```
Transformations (RDD lineage graph)

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
val warningsRDD = inputRDD.filter(line => line.contains("warning"))
val badLinesRDD = errorsRDD.union(warningsRDD)
```

Actions [1/2]

- Returns a final value to the driver program
- Or writes data to an external storage system

```
println(" Input had "+ badLinesRDD.count() + " concerning lines")
println(" Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

• Returns a final value to the driver program
• Or writes data to an external storage system
• Log file analysis example is continued
• `take()` Retrieves a small number of elements in the RDD at the driver program
• Iterates over them locally to print out information at the driver

Benefits of RDDs as a distributed memory abstraction [1/3]

- RDDs can only be created ("written") through coarse-grained transformations
- Distributed shared memory (DSM) allows reads and writes to each memory location
- Reads on RDDs can still be fine-grained
- A large read-only lookup table
- Applications perform bulk writes
- More efficient fault tolerance
- Lineage based bulk recovery

Benefits of RDDs as a distributed memory abstraction [2/3]

- RDDS' immutable data
- System can mitigate slow nodes (Stragglers)
- Creates backup copies of slow tasks
  - without accessing the same memory
- Spark distributes the data over different working nodes that run computations in parallel
  - Orchestrates communicating between nodes to integrate intermediate results and combine them for the final result

Benefits of RDDs as a distributed memory abstraction [3/3]

- Runtime can schedule tasks based on data locality
  - To improve performance
- RDDs degrade gracefully when there is insufficient memory
  - Partitions that do not fit in the RAM are stored on disk