PART A. BIG DATA TECHNOLOGY
3. DISTRIBUTED COMPUTING MODELS FOR SCALABLE BATCH COMPUTING
SECTION 1: MAPREDUCE

FAQs
- PA1
  - Your port ranges are announced in canvas

Topics of Today's Class
- 3. Distributed Computing Models for Scalable Batch Computing
  - MapReduce – II
  - Introduction to Spark
  - Quiz 1

MapReduce Data Flow

MapReduce data flow with a single reducer

MapReduce data flow with multiple reducers

http://www.cs.colostate.edu/~cs535
How MapReduce Works

Programming components of MapReduce
- Driver
- Mapper
- Reducer
- InputFormat
- Combiner
- Partitioner
- OutputFormat

Data locality optimization
- Hadoop tries to run the map task on a node where the input data resides in HDFS
  - Minimizes usage of cluster bandwidth
- If all replication nodes are running other map tasks
  - The job scheduler will look for a free map slot on a node in the same rack

Shuffle
- The process by which the system performs the sort and transfers the map outputs to
  the reducers as inputs
  - MapReduce guarantees that the input to every reducer is sorted by key
Combiner functions

- Minimize data transferred between map and reduce tasks
- Users can specify a *combiner function*
  - To be run on the map output
  - To replace the map output with the combiner output

Combiner example

- Example (from the previous max temperature example)—Without combiner
  - The first map produced,
    - (1950, 0), (1950, 20), (1950, 10)
  - The second map produced,
    - (1950, 25), (1950, 15)
  - The reduce function is called with a list of all the values,
    - (1950, [0, 20, 10, 25, 15])
  - Output will be,
    - (1950, 25)

We may express the function as,

\[
\text{max}(0, 20, 10, 25, 15) = \text{max}(\text{max}(0, 20, 10), \text{max}(25, 15)) \\
= \text{max}(20, 25) = 25
\]

Combiner example

- Example (from the previous max temperature example)—With combiner
  - The first map produced,
    - (1950, 0), (1950, 20), (1950, 10) → (1950, 20)
  - The second map produced,
  - The reduce function is called with a list of all the values,
    - (1950, [20, 25])
  - Output will be,
    - (1950, 25)

Combiner function

- Run a *local* reducer over Map output

- Reduce the amount of data shuffled between the mappers and the reducers
- Combiner cannot replace the reduce function
- Why?

Combiner function: Requirements

- Function should be commutative and associative
- Finding Maximum number
- Finding distribution
- Calculating Sum
- Finding an average

Combiner function: Requirements

- Function should be commutative and associative
- Finding Maximum number (YES)
- Finding distribution (YES)
- Calculating Sum (YES)
- Finding an average (YES/NO): if your combiner deliver the count of items, it is still possible.
YARN Framework

YARN (MapReduce 2)
- To provide the scalability to MapReduce
  - Splitting responsibility of the jobtracker
  - Scheduling
  - Task progress monitoring
- MapReduce is one type of YARN application

YARN (MapReduce 2)
- Resource manager
  - Manages the use of resources across the cluster
- Node manager
  - Launches and monitors the compute containers on machines in the cluster
- Application master
  - Manages the lifecycle of applications running on the cluster
    - Number of container and certain memory limit
    - Node managers oversee containers not to use more resources than allocated

A MapReduce job using YARN

Progress and status updates
- Task reports its progress and status back to its application master
  - Every 3 seconds over the umbilical interface
- The client polls the application master every second
  - mapreduce.client.progressmonitor.pollinterval

http://www.cs.colostate.edu/~cs535
In-Memory Cluster Computing: Apache Spark

Introduction

This material is built based on

- Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," The 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)


- Spark programming guide
  - https://spark.apache.org/docs/2.2.0/
  - Job Scheduling
    - https://spark.apache.org/docs/2.2.0-preview/join-scheduling.html

Distributed processing with the Spark framework

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

Inefficiencies for emerging applications: (2) Interactive data analytics

- User runs multiple ad-hoc queries on the same subset of the data

http://www.cs.colostate.edu/~cs535  Spring 2019 Colorado State University
### 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 */
object SimpleApp {
  def main(args: Array[String]): Unit = {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system
    val spark = new SparkSession builder .appName("Simple Application") .getOrCreate() .master("local") .getOrCreate()

    val numAs = logFile .lines .filter(_.contains("a")).count() .println(numAs)

    val numBs = logFile .lines .filter(_.contains("b")).count() .println(numBs)

    println(s"Lines with a: $numAs, Lines with b: $numBs")
  }
}
```

### Scala build and run

```scala
// Your directory layout should look like this
$ cd -
$ sbt
$ .sbt
$ s瑛/scala
$ .s瑛/scala/simpleApp.scala
# Package a jar containing your application
$ sbt package
# (SBT) Packaging ...
$ slant/s瑛/scala-2.11/simple-project_2.11-1.0.jar
$ slant/s瑛/scala-2.11/simple-project_2.11-1.0.jar
# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit 
  --class "SimpleApp" 
  master "local[4]" 
  target/s瑛/scala-2.11/simple-project_2.11-1.0.jar
  target/s瑛/scala-2.11/simple-project_2.11-1.0.jar
```

---

http://www.cs.colostate.edu/~cs535

Spring 2019 Colorado State University
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

Creating RDDs [1/3]

- Loading an external dataset
  ```scala
  val lines = sc.textFile("/path/to/README.md")
  ```
- Parallelizing a collection in your driver program
  ```scala
  val lines = sc.textFile("/path/to/README.md")
  ```

Creating RDDs [2/3]

- Line 1: defines a base RDD from an external file
  - This dataset is not loaded in memory
- Line 2: defines `lineLengths` as the result of map transformation
  - It is not immediately computed
- Line 3: performs reduce and compute the results

Creating RDDs [3/3]

- If you want to use `lineLengths` again later

Spark Programming Interface to RDD [1/3]

- Transformations
  - Operations that create RDDs
    - Return pointers to new RDDs
      - e.g. map, filter, and join
  - RDDs can only be created through deterministic operations on either
    - Data in stable storage
    - Other RDDs

http://www.cs.colostate.edu/~cs535
**Spark Programming Interface to RDD**

- **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

```scala
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

- **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*

```scala
lineLengths.persist()
```

**Questions?**