Big Data Frameworks

• A system that allows developers to write a program and execute it on a cluster of machines.
• Hides most of the low-level system issues such as fault tolerance, network communication, and load balancing.
• Imposes some restrictions on the developer to ensure that they can run the program efficiently.
I often repeat repeat myself
I often repeat repeat
I don’t don’t know why know why
I simply know that I I I
am am inclined to say to say
a lot a lot this way this way
I often repeat repeat myself
I often repeat repeat

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>am</td>
<td>2</td>
</tr>
<tr>
<td>don't</td>
<td>2</td>
</tr>
<tr>
<td>I</td>
<td>9</td>
</tr>
<tr>
<td>inclined</td>
<td>1</td>
</tr>
<tr>
<td>know</td>
<td>3</td>
</tr>
<tr>
<td>lot</td>
<td>2</td>
</tr>
<tr>
<td>myself</td>
<td>2</td>
</tr>
<tr>
<td>often</td>
<td>4</td>
</tr>
<tr>
<td>repeat</td>
<td>8</td>
</tr>
<tr>
<td>say</td>
<td>2</td>
</tr>
<tr>
<td>simply</td>
<td>1</td>
</tr>
<tr>
<td>that</td>
<td>1</td>
</tr>
<tr>
<td>this</td>
<td>2</td>
</tr>
<tr>
<td>to</td>
<td>2</td>
</tr>
<tr>
<td>way</td>
<td>2</td>
</tr>
<tr>
<td>why</td>
<td>2</td>
</tr>
</tbody>
</table>
Word Count Walkthrough (1/2)

Input file splits

Line records

Word-count pairs
Word Count Walkthrough (2/2)

Partition #3.1
- don't 2
- know 2
- myself 1
- often 2
- repeat 4
- why 2

Partition #3.2
- a 2
- am 2
- inclined 1
- know 1
- lot 2
- say 2
- simply 1
- that 1
- to 2
- this 2
- way 2

Partition #3.3
- I 2
- myself 1
- often 2
- repeat 4

Partition #4.1
- a 2
- am 2
- inclined 1
- know 1
- know 2
- myself 1
- myself 1
- often 2
- say 2
- simply 1
- way 2
- why 2

Partition #4.2
- don't 2
- lot 2
- repeat 4
- that 1
- this 2
- to 2

Partition #5.1
- a 2
- am 2
- I 9
- inclined 1
- know 3
- myself 2
- often 4
- say 2
- simply 1
- way 2
- why 2

Partition #5.2
- don't 2
- lot 2
- repeat 8
- that 1
- this 2
- to 2

Partial word count
Shuffled records
Final word count
Output
Word Count Logic

• The logic behind the word count example can be expressed using only two functions
  ▪ WordExtractor: String → {(w, 1)}
  ▪ WordSum: (w, {c}) → (w, Σc)
Complete Word Count in Hadoop

public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable>{
private final static IntWritable one = new IntWritable(1);
private Text word = new Text();
public void map(Object key, Text value, Context context)
    {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}

public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
private IntWritable result = new IntWritable();
public void reduce(Text key, Iterable<IntWritable> values, Context context)
    {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}

Source: https://hadoop.apache.org/docs/r3.2.2/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0
Complete Word Count in Spark

// In Scala shell
val lines = sc.textFile("data.txt")
val pairs = lines.flatMap(s => s.split("\b")).map(w => (w,1))
val counts = pairs.reduceByKey((a, b) => a + b)
counts.saveAsTextFile("word_count_output.txt")
Big-data Processing

• Programming Model
  ▪ How a developer writes a big-data program

• Application Model (Logical Model)
  ▪ How the big-data platform internally represents a user program

• Execution Model (Physical Model)
  ▪ How the program gets executed on the cluster
Functional Programming Model

• A big-data program consists of user-defined functions
  ▪ E.g., Map and Reduce functions in a Hadoop MapReduce program

• A valid function must satisfy two constraints
  ▪ Stateless/Memoryless
  ▪ Deterministic
Functional Programming

• Which of these are functions?
Functional Programming

• Which of these are functions?

A  B  C  D

input

output
Word Count Functions

• Word Extractor(Line: String) {
  words = Line.split
  foreach (w ∈ words) output.write(w, 1)
}

• SumWords(word: String, counts: int[]) {
  sum = sum(counts)
  output.write(word, sum)
}
Examples

Function1(x) {
    return x + 5;
}

RNG random;
Function3(x) {
    random.randomInt(0, x);
}

Map<String, Int> lookuptable;
Function4(x) {
    return lookuptable.get(x);
}

Int sum
Function2(x) {
    sum += x;
    return sum;
}
Examples

Function1(x) {
    return x + 5;
}

Int sum
Function2(x) {
    sum += x;
    return sum;
}

RNG random;
Function3(x) {
    random.randomInt(0, x);
}

Map<String, Int> lookuptable;
Function4(x) {
    return lookuptable.get(x);
}
Directed Acyclic Graph

• The functional programming paradigm allows the developer to define one function
• The program consists of multiple functions
// In Scala shell
val lines = sc.textFile("data.txt")
val pairs = lines.flatMap(s => s.split("\\b")).map(w => (w, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
counts.saveAsTextFile("word_count_output.txt")
Bulk Synchronous Parallel (BSP)

• The BSP model is how big-data frameworks execute a program
• The model splits the execution into stages of local processing
• Computation stages are separated by a communication barrier
BSP Model

Processor 1
Independent Local Processing

Processor 2
Independent Local Processing

...  

Processor n
Independent Local Processing

Communication

Barrier
Communication Patterns

Fully Connected
(Requires network communication)

One-to-one
(Can be done locally in one stage)
Word Count Stages

Stage 1
- input
  - flatMap(...)
- words
  - map(...)
Architecture of Big-data Frameworks

User → Driver Node → Master Node → Executor Nodes

Main program

Job DAG + Functions + Configuration

Tasks for stages
Hadoop MapReduce

Input

$k_1, v_1$

$k_1, v_1$

...

$k_1, v_1$

Map

Intermediate Data

$k_2, v_2$

$k_2, v_2$

...

$k_2, v_2$

Reduce

Output

$k_3, v_3$

$k_3, v_3$

...

$k_3, v_3$
Map and Reduce Functions

• Map Function
  ▪ Maps a single input record to a set (possibly empty) of intermediate records
  ▪ Map: \( \langle k_1, v_1 \rangle \rightarrow \{ \langle k_2, v_2 \rangle \} \)

• Reduce Function
  ▪ Reduces a set of intermediate records with the same key to a set (possibly empty) of output records
  ▪ Reduce: \( \langle k_2, \{ v_2 \} \rangle \rightarrow \{ \langle k_3, v_3 \rangle \} \)

• Combine Function (Optional)
  ▪ Partial local reduction before reduce
  ▪ Combine: \( \langle k_2, \{ v_2 \} \rangle \rightarrow \{ \langle k_2, v_2 \rangle \} \)
MapReduce DAG

Input

Map output

Combine Output

Reduce Output

Map Stage

Reduce Stage
MapReduce Program Cycle

User → Driver Node → Master Node → Executor Nodes

Main program

JAR File + Configuration

Map and Reduce Tasks
Job Execution Overview

Job submission
Job preparation
Map
Shuffle
Reduce
Cleanup
Job Submission

• Execution location: Driver node
• A driver machine should have the following
  ▪ Compatible Hadoop binaries
  ▪ Cluster configuration files
  ▪ Network access to the master node
• Collects job information from the user
  ▪ Input and output paths
  ▪ Map, reduce, and any other functions
  ▪ Any additional user configuration
• Packages all this in a Hadoop Configuration
Hadoop Configuration

<table>
<thead>
<tr>
<th>Key: String</th>
<th>Value: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>hdfs://user/eldawy/README.txt</td>
</tr>
<tr>
<td>Output</td>
<td>hdfs://user/eldawy/wordcount</td>
</tr>
<tr>
<td>Mapper</td>
<td>edu.ucr.cs.cs167.eldawy.WordCount</td>
</tr>
<tr>
<td>Reducer</td>
<td>...</td>
</tr>
<tr>
<td>JAR File</td>
<td>...</td>
</tr>
<tr>
<td>User-defined</td>
<td>User-defined</td>
</tr>
</tbody>
</table>

Serialized over network

Master node
Job Preparation

• Runs on the master node
• Gets the job ready for parallel execution
• Collects the JAR file that contains the user-defined functions, e.g., Map and Reduce
• Writes the JAR and configuration to HDFS to be accessible by the executors
• Looks at the input file(s) to decide how many map tasks are needed
• Makes some sanity checks
• Finally, it pushes the BRB (Big Red Button)
Job Preparation

Configuration

Master node

HDFS

JAR File

InputFormat#getSplits()
Map Phase

• Runs in parallel on worker nodes

• $M$ Mappers:
  ▪ Read the input
  ▪ Apply the map function
  ▪ Apply the combine function (if configured)
  ▪ Store the map output

• There is no guaranteed ordering for processing the input splits
Input record reader

• Split the file based on the file metadata
  ▪ File size, block sizes, # of nodes
• Each split is defined by:
  ▪ File name, Start offset, Length
• For each split:
  ▪ Seek to the start offset
  ▪ Skip the first record (except for the first split)
  ▪ Read until the beginning of the record goes beyond the start + length
Map Phase

Master node

Input Splits (Map tasks)

\[ IS_1 \quad IS_2 \quad IS_3 \quad IS_4 \quad IS_5 \quad \ldots \quad IS_M \]
Map Task

• Reads the job configuration and task information (mainly, InputSplit)
• Instantiates an object of the Mapper class
• Instantiates a record reader for the assigned input split
• Reads records one-by-one from the record reader and passes them to the user-defined map function
• Passes the map output to the next step
Map output

• What happens to the map output?
• It depends on the number of reducers
  • 0 reducers: Map output is written directly to HDFS as the final answer
  • 1+ reducers: Map output is passed to the shuffle phase
Shuffle Phase

• Executed only in the case of one or more reducers
• Transfers data between the mappers and reducers
• Groups records by their keys to ensure local processing in the reduce phase
Shuffle Phase

Map_1 \rightarrow \text{Reduce}_1
Map_2 \rightarrow \text{Reduce}_2
Map_3 \rightarrow \text{Reduce}_2
\ldots
Map_M \rightarrow \text{Reduce}_N
Shuffle Phase (Map-side)

Input Split → Map

Partition

Reduce$_1$  Reduce$_2$  ...  Reduce$_N$

Map$_i$
Shuffle Phase (Reduce-side)

Map\(_1\) \rightarrow \text{part}_1
\rightarrow \text{Reduce}_j
\rightarrow k\ v
\rightarrow \text{Sort}
\rightarrow \text{Copy}

Map\(_2\) \rightarrow \text{part}_2

Map\(_3\) \rightarrow \text{part}_3

…

Map\(_M\) \rightarrow \text{part}_M

k\ v

k\ v

k\ v

k\ v

k\ v

k\ v

k\ v

k\ v
Reduce Phase

• Apply the reduce function to each group of similar keys
Output Writing

• Materializes the final output to disk
• All results are from one process (mapper/reducer) are stored in a subdirectory
• An OutputFormat is used to
  ▪ Create any files in the output directory
  ▪ Write the output records one-by-one to the output
  ▪ Merge the results from all the tasks (if needed)
• While the output writing runs in parallel, the final commit step runs on a single machine
Hadoop MapReduce Conclusion

• A MapReduce program consists of a map, a reduce, and optionally a combine function
• Hadoop distributes the program to all executor nodes
• The input is partitioned and each input split is processed independently
• The intermediate data is shuffled and reduced to produce the final output.
Spark Resilient Distributed Dataset (RDD)
• A memory-based big-data framework
• Resilient Distributed Dataset (RDD) is an alternative query processing framework for big-data
• Utilizes more memory to speed up query processing
RDD Abstraction

• RDD is a pointer to a distributed dataset
• Stores information about how to compute the data or where the data is
• Transformation: Converts an RDD to another RDD
• Action: Returns an answer of an operation over an RDD
RDD

Operation

RDD → RDD
Filter Operation

Similar to the map operation

Filter
GroupBy (Shuffle) Operation

Similar operation Join
Application DAG

• A complete DAG consists:
  ▪ One or more input loader
  ▪ Zero or more transformations
  ▪ One action
DAG Execution using BSP

How does Spark split a DAG into stages?
Types of Dependencies

- Narrow dependencies
- Wide dependencies

Narrow dependencies:
Each partition of the parent RDD is used by at most one partition of the child RDD.

Wide dependencies:
Each partition of the parent RDD may be depended on by multiple child partitions.

Source: https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies
Spark RDD Features

• Lazy execution: Collect transformations and execute on actions
• Lineage tracking: Keep track of the lineage of each RDD for fault-tolerance
• Resiliency: When an in-memory partition gets lost, Spark recomputes it
Examples of Transformations

• map
• mapToPair
• flatMap
• reduceByKey
• filter
• sample
• join
• union
• partitionBy
Examples of Actions

• count
• collect
• save(path)
• persist
• reduce
RDD Operations

• Spark is richer than Hadoop in terms of operations
• Sometimes, you can do the same logic with more than one way
• In the following part, we will explain how different RDD operations work
• The goal is to understand the performance implications of these operations and choose the most efficient one
Java Examples

- Apache Spark homepage
  - https://spark.apache.org

```java
// Initialize the Spark context
JavaSparkContext spark =
    new JavaSparkContext("local", "CS167-Demo");
```
# Initialize the Spark context

JavaSparkContext spark =
    new JavaSparkContext("local", "CS167-Demo");

# Hello World! Example. Count the number of lines in the file

JavaRDD<String> textFileRDD =
    spark.textFile("nasa_19950801.tsv");

long count = textFileRDD.count();
System.out.println("Number of lines is "+count);
# Count the number of OK lines (response code 200)

```java
JavaRDD<String> okLines = textFileRDD.filter(new Function<String, Boolean>() {
    @Override
    public Boolean call(String s) throws Exception {
        String code = s.split("\t")[5];
        return code.equals("200");
    }
});
long count = okLines.count();
System.out.println("Number of OK lines is "+count);
```
# Count the number of OK lines (response code 200)
# Shorten the implementation using lambdas (Java 8 and above)

```java
JavaRDD<String> okLines =
textFileRDD.filter(s -> s.split("\t")[5].equals("200"));

long count = okLines.count();
System.out.println("Number of OK lines is "+count);
```
# Make it parametrized by taking the response code as a command line argument

String inputFileName = args[0];
String desiredResponseCode = args[1];

...  
JavaRDD<String> textFileRDD = spark.textFile(inputFileName);
JavaRDD<String> okLines = textFileRDD.filter(new Function<String, Boolean>() {
    @Override
    public Boolean call(String s) {
        String code = s.split("\t")[5];
        return code.equals(desiredResponseCode);
    }
});
Examples

# Count by response code

# Important! Not all transformations and actions are on the getting started guide

JavaPairRDD<Integer, String> linesByCode = textFileRDD.mapToPair(new PairFunction<String, Integer, String>() {
    @Override
    public Tuple2<Integer, String> call(String s) {
        String code = s.split("\t")[5];
        return new Tuple2<Integer, String>(Integer.valueOf(code), s);
    }
});

Map<Integer, Long> countByCode = linesByCode.countByKey();
System.out.println(countByCode);
RDD<T>#filter

- `func: T → Boolean`
- Applies the predicate function on each record and produces that tuple only if the predicate returns true
- Result RDD<T> with same or fewer records than the input
- In Hadoop:
  - `map(T value) {
      if (func(value))
        context.write(value)
    }
`
**RDD<T>#map(func)**

- **func**: $T \rightarrow U$
- Applies the map function to each record in the input to produce one record
- Results in RDD<U> with the same number of records as the input
- **In Hadoop:**
  ```java
  map(T value) {
    context.write(func(value));
  }
  ```
RDD<T>flatMap(func)

• func: T → Iterator<V>
• Applies the map function to each record and add all resulting values to the output RDD
• Result: RDD<V>
• This is the closest function to the Hadoop map function
• In Hadoop:
  ▪ map(T value) {
    Iterator<V> results = func(value);
    for (V result : results)
      context.write(result)
  }
RDD<T>#mapPartition(func)

- **func**: Iterator<T> → Iterator<U>
- Applies the map function to a list of records in one partition in the input and adds all resulting values to the output RDD
- Can be helpful in two situations
  - If there is a costly initialization step in the function
  - If many records can result in one record
- **Result**: RDD<U>
RDD<T>\#mapPartition(func)

- In Hadoop, the mapPartition function can be implemented by overriding the run() method in the Mapper, rather than the map() function

```java
run(context) {
    // Initialize
    Array<T> values;
    for (T value : context)
        values.add(value);
    Iterator<V> results = func(values);
    for (V value : results)
        context.write(value);
}
```
RDD<T>:#mapPartitionWithIndex(func)

• func: (Integer, Iterator<T>) → Iterator<U>
• Similar to mapPartition but provides a unique index for each partition
• In Hadoop, you can achieve a similar functionality by retrieving the InputSplit or taskID from the context.
RDD<T> #sample(r, f, s)

- r: Boolean: With replacement (true/false)
- f: Float: Fraction [0,1]
- s: Long: Seed for random number generation

- Returns RDD<T> with a sample of the records in the input RDD
- Can be implemented using mapPartitionWithIndex as follows
  - Initialize the random number generator based on seed and partition index
  - Select a subset of records as desired
  - Return the sampled records
RDD<T>#distinct()

• Removes duplicate values in the input RDD
• Returns RDD<T>
• Implemented as follows
  map(x => (x, null)).
  reduceByKey((a, b) => a, numPartitions).
  map(_.1)
• Note: Both a and b are null in the reduceByKey function above
RDD<T>#reduce(func)

- func: (T, T) → T
- This is not the same as the reduce function of Hadoop even though it has the same name
- Reduces all the records to a single value by repeatedly applying the given function
- Result: T
- This is an action
• In Hadoop
  ▪ map(T value) {
    context.write(NullWritable.get(), value);
  }
  ▪ combine, reduce(key, Iterator<T> values) {
    T result = values.next();
    while (values.hasNext())
      result = func(result, values.next());
    context.write(result);
  }
RDD<T>#reduce(func)

Partition #1: Local Processing

Partition #2: Local Processing

Driver Machine

Final Result

Network Transfer

Local Processing

P#3

P#4
RDD<\textit{K,V}> \#reduceByKey(func)

- func: (V, V) \rightarrow V
- Similar to reduce but applies the given function to each group separately
- Since there could be so many groups, this operation is a transformation that can be followed by further transformations and actions
- Result: RDD<\textit{K,V}>
- By default, number of reducers is equal to number of input partitions but can be overridden
RDD\langle K, V \rangle \#reduceByKey(func)

• In Hadoop:
  ▪ map(K key, V value) {
    context.write(key, value);
  }
  ▪ combine, reduce(K key, Iterator<V> values) {
    V result = values.next();
    while (values.hasNext())
      result = func(result, values.next());
    context.write(key, result);
  }
Limitation of reduce methods

• Both reduce methods have a limitation is that they have to return a value of the same type as the input.

• Let us say we want to implement a program that operates on an RDD<Integer> and returns one of the following values
  ▪ 0: Input is empty
  ▪ 1: Input contains only odd values
  ▪ 2: Input contains only even values
  ▪ 3: Input contains a mix of even and odd values
RDD<T>#aggregate(zero, seqOp, combOp)

- zero: U - Zero value of type U
- seqOp: (U, T) → U – Combines the aggregate value with an input value
- combOp: (U, U) → U – Combines two aggregate values
- Returns U

- Similarly, aggregateByKey operates on RDD<K,V> and returns RDD<K,U>
\textbf{RDD}\langle T\rangle \text{#} \text{aggregate}(\text{zero, seqOp, combOp})

- In Hadoop:
  - \texttt{run(context) \{}
    \begin{itemize}
    \item \texttt{U result = zero;}
    \item \texttt{for (T value : context)}
      \begin{itemize}
      \item \texttt{result = seqOp(result, value);}
      \item \texttt{context.write(NullWritable.get(), result);}
      \end{itemize}
    \end{itemize}
  \}

  \texttt{combine,reduce(key, Iterator\langle U \rangle values) \{}
  \begin{itemize}
  \item \texttt{U result = values.next();}
  \item \texttt{while (values.hasNext())}
    \begin{itemize}
    \item \texttt{result = combOp(result, values.next());}
    \item \texttt{context.write(result);}
    \end{itemize}
  \end{itemize}
\}
$\text{RDD}\langle T \rangle \# \text{aggregate}(\text{zero, seqOp, combOp})$

- Example:
- $\text{RDD}\langle \text{Integer} \rangle$ values
- Byte marker = values.aggregate( (Byte)0,
  (result: Byte, x: Integer) => {
    if (x % 2 == 0) // Even
      return result | 2;
    else
      return result | 1;
  },
  (result1: Byte, result2: Byte) => result1 | result2
);
RDD<T>#aggregate(zero, seqOp, combOp)
RDD<K,V> #groupByKey()

• Groups all values with the same key into the same partition
• Closest to the shuffle operation in Hadoop
• Returns RDD<K, Iterator<V>>
• Performance notice: By default, all values are kept in memory so this method can be very memory consuming.
• Unlike the reduce and aggregate methods, this method does not run a combiner step, i.e., all records get shuffled over network
Further Readings

• List of common transformations and actions

• Spark RDD Scala API