Spark RDD Operations
Transformations and Actions
RDD Processing Model

• RDD can be modeled using the Bulk Synchronous Parallel (BSP) model

Processor 1

Independent Local Processing

Processor 2

Independent Local Processing

...  

Processor n

Independent Local Processing

Communication

Barrier
• In Spark RDD, you can generally think of these two rules
  ▪ Narrow dependency ➔ Local processing
  ▪ Wide dependency ➔ Network communication
A simple abstraction for local processing
Based on functional programming
Local Processing(input: Iterator<T>, output: Writer<U>) {
  ...
  // output.write(U)
}
Functional Programming

• RDD is a functional programming paradigm

• Which of these are functions?
Functional Programming

• RDD is a functional programming paradigm

• Which of these are functions?
Function Limitations

• For one input, the function should return one output
• The function should be memoryless
  ▪ Should not remember past input
• The function should be stateless
  ▪ Should not change any state when called
• It is up to the developer to enforce these properties
Examples

Function 1(x) {
    return x + 5;
}

Int sum
Function 2(x) {
    sum += x;
    return sum;
}

RNG random;
Function 3(x) {
    random.randomInt(0, x);
}

Map<String, Int> lookuptable;
Function 4(x) {
    return lookuptable.get(x);
}
Examples

Function 1(x) {
    return x + 5;
}

Int sum
Function 2(x) {
    sum += x;
    return sum;
}

RNG random;
Function 3(x) {
    random.randomInt(0, x);
}

Map<String, Int> lookuptable;
Function 4(x) {
    return lookuptable.get(x);
}
Network Communication

• a.k.a. shuffle operation
• Given a record $r$ and $n$ partitions:
  ▪ Assign the record to one of the partitions $[0, n - 1]$
RDD Operations

• Spark is rich with 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
RDD\<T\>#filter

• Filter is a function\<T \rightarrow 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
• Local Processing {
  for-each (t in input) {
    if (func(t))
      output.write(t)
  }
}
RDD<T>#map(func)

- func: T → 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
- Local Processing {
  for-each (t in input) 
    output.write(func(t))
}
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
- **Local Processing**
  ```java
  Iterator<V> results = func(input);
  for (V result : results)
    output.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>
• Local Processing {
    results = func(input)
    for-each (v in results)
        output.write(v);
}
RDD<T>#mapPartitionWithIndex(func)

• func: (Integer, Iterator<T>) → Iterator<U>
• Similar to mapPartition but provides a unique index for each partition
• To achieve this in Spark, the partition ID is passed to the function
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>#reduce(func)

- func: (T, T) → T
- Reduces all the records to a single value by repeatedly applying the given function
- The function should be associative and commutative
- Result: T
- This is an action
RDD<T> #reduce(func)

• mapPartition {
    T result = input.next
    for-each (r in input)
        result = reduce(result, r)
    return result
}

• Shuffle: assign all records to one partition

• Collect partial results and apply the same function again
RDD<T>#reduce(func)
RDD<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<K, V>
- By default, number of reducers is equal to number of input partitions but can be overridden
RDD<K,V>#$reduceByKey(func)

- mapPartition {
  Map<K,V> results;
  for-each ((k,v) in input) {
    if (results.contains(k))
      results[k] = reduce(results[k], v);
    else
      results[k] = v;
  }
}

- Shuffle by key, assign (k,v) to hash(k) mod n

- mapPartition {
  // All input records have the same key
  V result = value.next
  for-each (v in values)
    result = reduce(result, v)
  output.write(k, v)
}
RDD<T>#distinct()

- Removes duplicate values in the input RDD
- Returns RDD<T>
- Implemented as follows
  map(x => (x, null)).
  reduceByKey((x, y) => x, numPartitions).
  map(_._1)
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
- Like reduce, aggregate is an action
- Returns U

- Similarly, aggregateByKey is a transformation that takes RDD<K,V> and returns RDD<K,U>
$\text{RDD}\langle T\rangle #$aggregate(zero, seqOp, combOp)

- **mapPartition** {
  
  U partialResult = zero
  for-each (t in input)
    result = seqOp(partialResult, t)
  return partialResult

- Collect all partial results into one partition

- **mapPartition** {
  
  U finalResult = input.next
  for-each (u in input)
    finalResult = combOp(finalResult, u)
  return finalResult

}
RDD<T>#aggregate(zero, seqOp, combOp)

- Example:
- RDD<Integer> values
- Byte marker = values.aggregate( (Byte)0,
  (result: Byte, x: Integer) => {
    if (x % 1 == 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
  ▪ http://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

• Spark RDD Scala API
  ▪ http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD