How Spark RDD Operations Work

Ahmed Eldawy

CELEBRATING 30 YEARS
Marlan and Rosemary Bourns
College of Engineering

Objectives

- Understand how several RDD operations internally work
- Map RDD operations to MapReduce
- Compare narrow dependency to wide dependency transformations implementation
- Assess the memory overhead of RDD functions

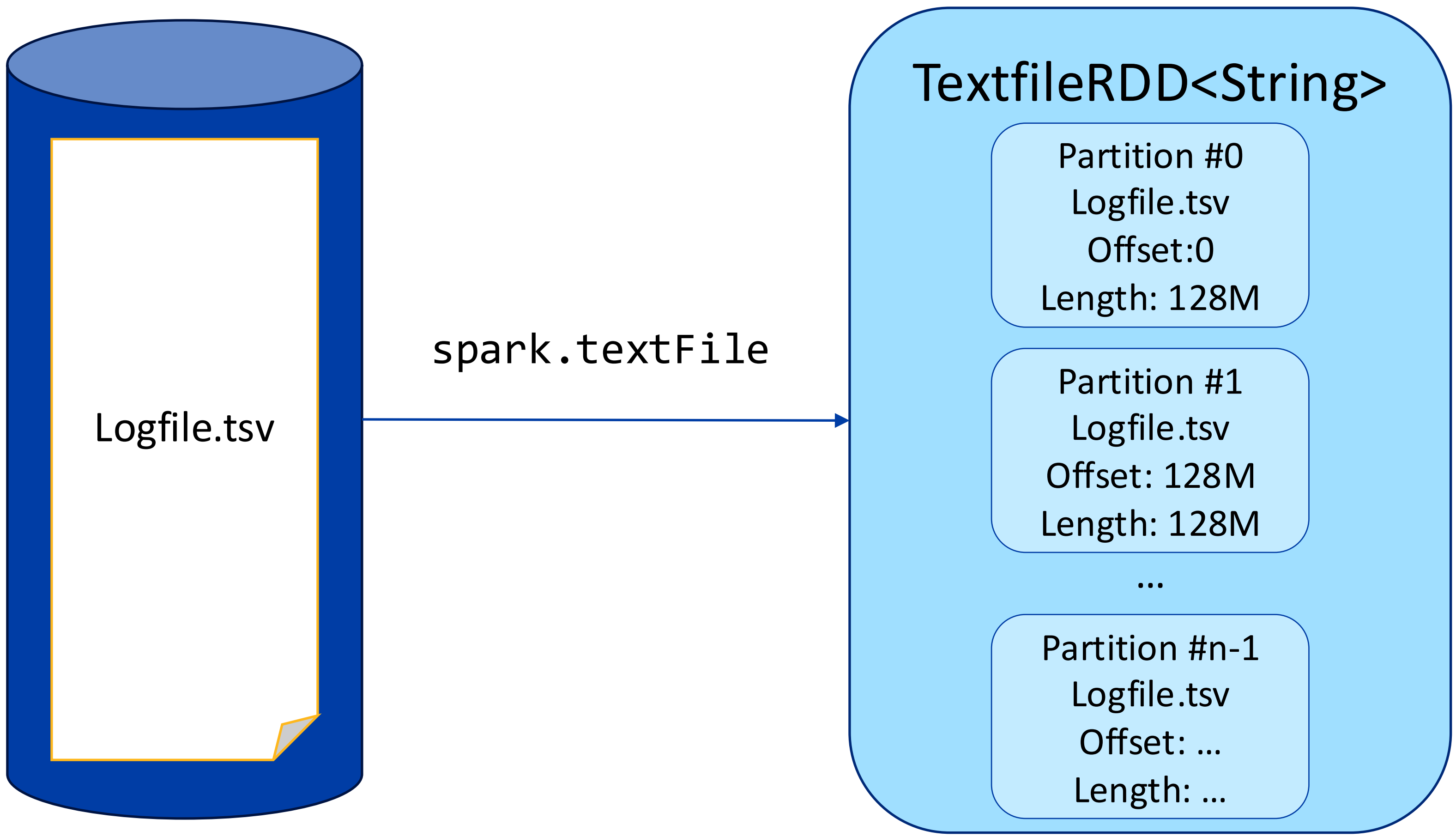
CELEBRATING 30 YEARS
Marlan and Rosemary Bourns College of Engineering

Spark Operations

- Data loader/creator. Creates the first RDD
- Transformation. Converts one RDD to another.
  - Narrow dependency
  - Wide dependency
- Action. Executes the application.

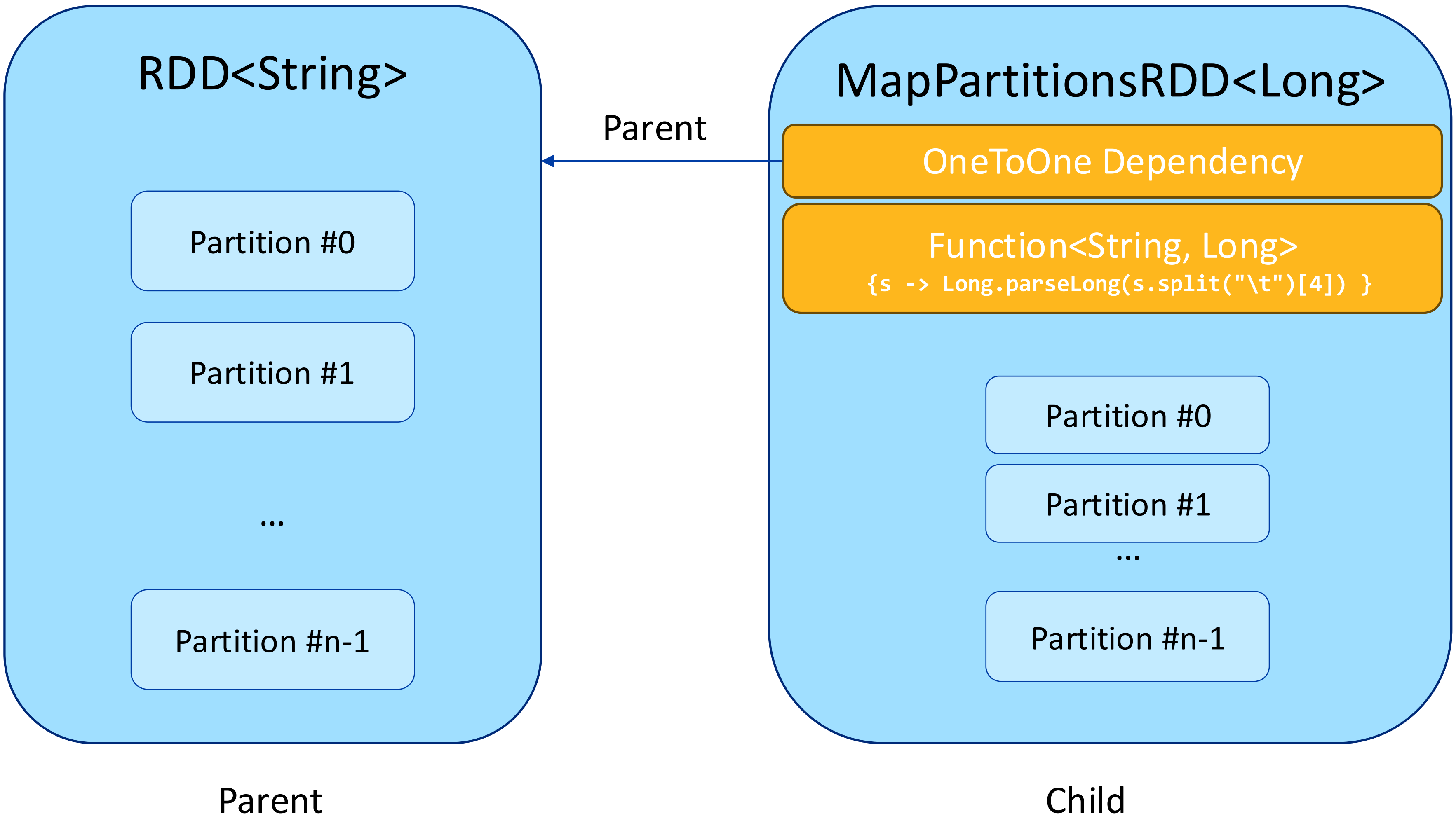
Data Loader

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



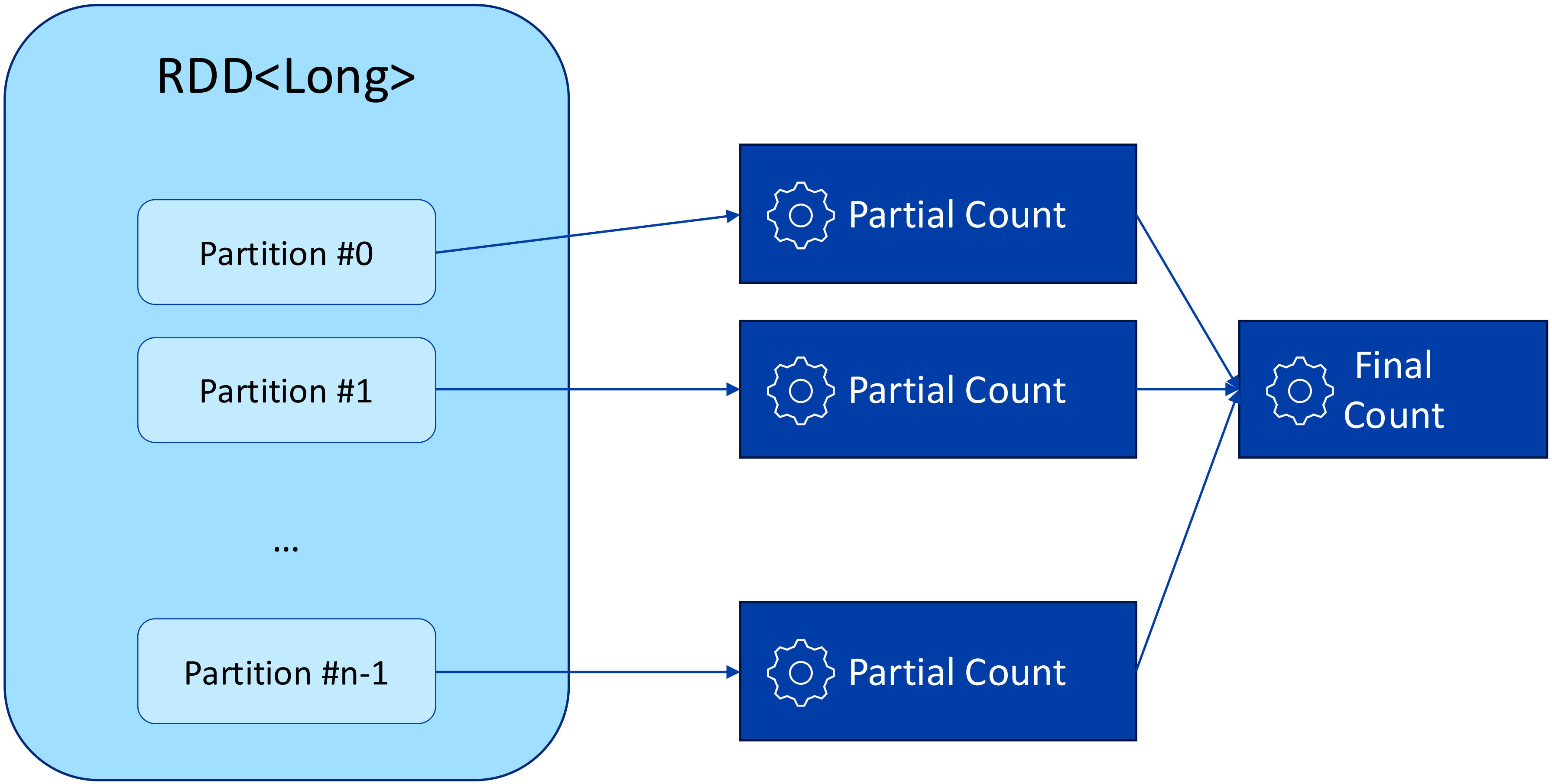
Map Operation

```java
JavaRDD<Long> sizes = textFileRDD.map(s ->
      Long.parseLong(s.split("\t")[4]));
```



Reduce Action

```
Long size = codes.reduce((a, b) -> a + b);
```



Hadoop Mapper (under the hood)


```
public void run(Context context) throws IOException, InterruptedException {
  setup(context);
  try {
    while (context.nextKeyValue()) {
      map(context.getCurrentKey(), context.getCurrentValue(), context);
    }
  } finally {
    cleanup(context);
  }
}
```

RDD Operation Notation

RDD<T>#operation(params, func): RDD<U>

- The operation is applied on an RDD with records of type T
- The output is an RDD with records of type U
- Types T and U can be any Java classes
- The operation can take additional parameters or functions
- The parameters can be any constant values
- func: {T → U}
  - indicates a user-defined function that takes an input of type T and return an output of type U

RDD<T>#filter(pred:{T→Boolean}): RDD<T>

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

RDD<T>#map(func: {T→U}): RDD<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
- Narrow dependency
- In Hadoop:
  - map(T value) {
      context.write(func.apply(value));
    }

RDD<T>#flatMap(func: {T→Iter<U>}): RDD<U>

- Applies the map function to each record and add all resulting values to the output RDD
- Result: RDD<U>
- This is the closest function to the Hadoop map function
- Narrow dependency
- In Hadoop:
  - ```
    map(T value) {
      Iterator<U> results = func.apply(value);
      for (U result : results)
        context.write(result)
    }
    ```
    ```

RDD<T>#mapPartition(func): RDD<U>

- 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
- Narrow dependency
- 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): RDD<U>

- In Hadoop, the mapPartition function can be implemented by overriding the run() method in the Mapper, rather than the map() function
  - ```
    run(context) {
      // Initialize
      Array<T> values;
      for (T value : context)
        values.add(value);
      Iterator<U> results = func(values.iterator());
      for (U value : results)
        context.write(value);
      // Cleanup
    }
    ```
    ```

RDD<T>#mapPartitionWithIndex(func): RDD<U>

- func: {(Integer, Iterator<T>) → Iterator<U>}
- Similar to mapPartition but provides a unique index for each partition
- Narrow dependency
- In Hadoop, you can achieve a similar functionality by retrieving the InputSplit or taskID from the context.

RDD<T>#sample(r: Boolean, f: Float, s: Long): RDD<T>

- 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
- Narrow dependency
- 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

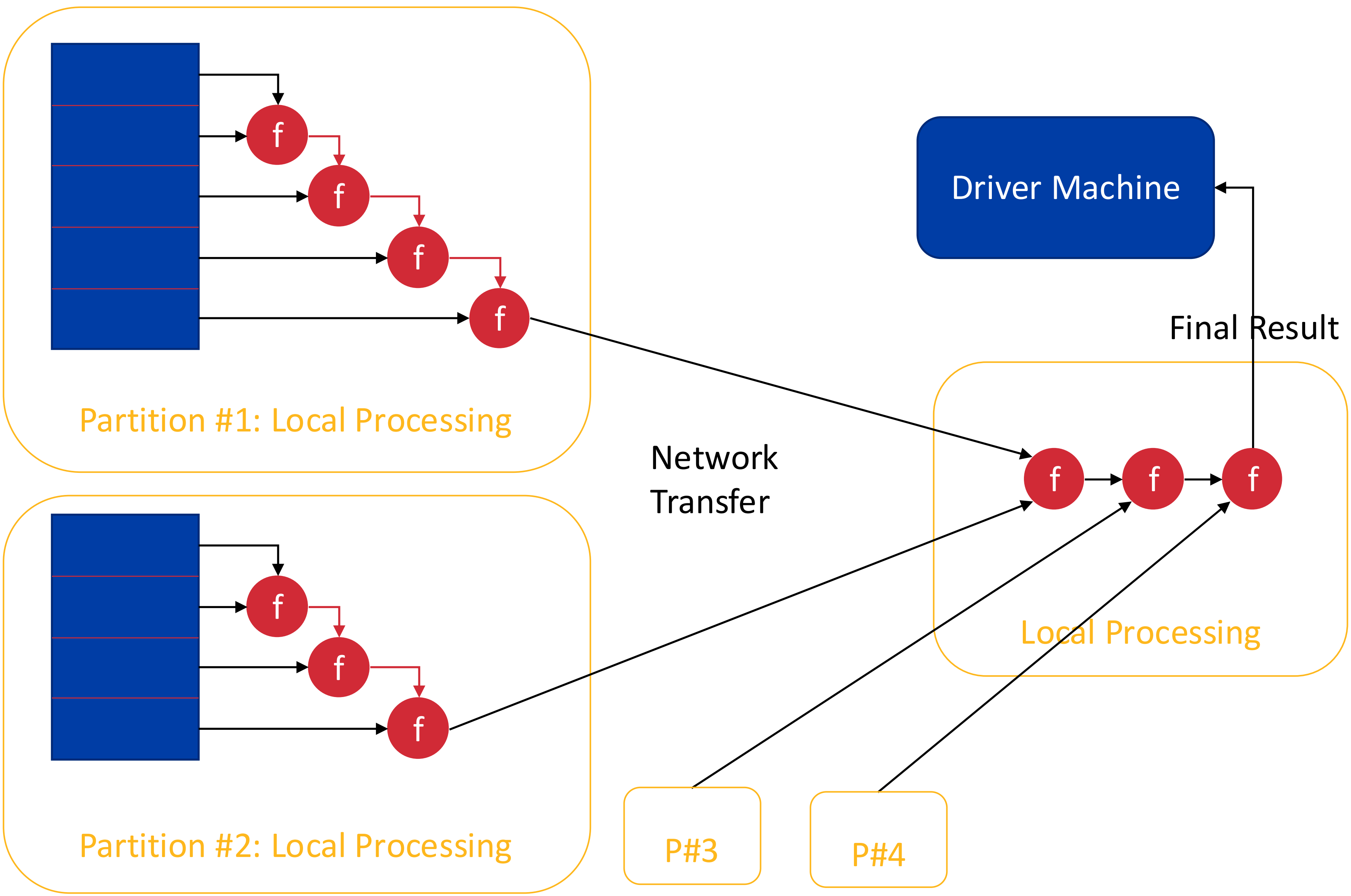
RDD<T>#reduce(func: {(T,T)→T}): T

- 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

RDD<T>#reduce(func: {(T,T)→T})

- 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: {(T,T)→T})



RDD<K,V>#reduceByKey(func: (V, V) → V): RDD<K,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
- Wide dependency
- 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)

- 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);
    }
    ```

RDD<T>#distinct(): RDD<T>

- 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
- Question: Is this a narrow or wide dependency transformation

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): U

- 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, hence, an action

- Similarly, aggregateByKey operates on RDD<K,V> and returns RDD<K,U>

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

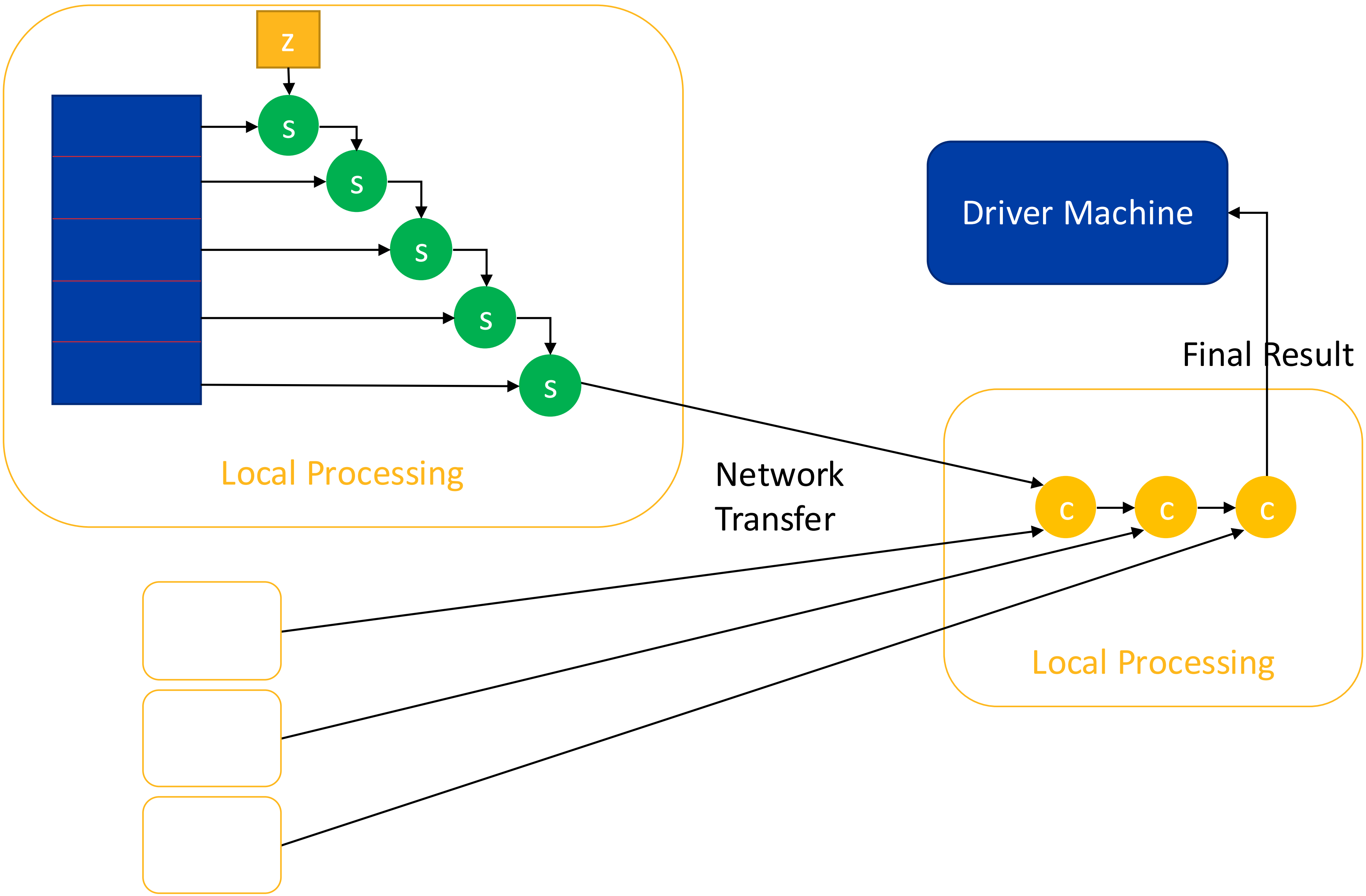
- In Hadoop:
  - run(context) {
    U result = zero;
    for (T value : context)
      result = seqOp(result, value);
    context.write(NullWritable.get(), result);
    }
  - combine,reduce(key, Iterator<U> values) {
    U result = values.next();
    while (values.hasNext())
      result = combOp(result, values.next());
    context.write(result);
    }

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

- Example:
- RDD<Integer> 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(): RDD<K, Iterator<V>>

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

RDD<T>#foreach(func: {T→None})

- An action that iterates over all records in parallel and applies a function on each one
- Notice: This given function runs in parallel on the worker nodes
- You cannot use this function to iterate over records on the local node

RDD<T>#collect(): Array[T]

- Returns the set of records in the RDD as an array
- This should only be used with very small datasets
- Spark has a default limit of 1GB for the total result size `(spark.driver.maxResultSize)`
- Related actions:
  - RDD<T>#take(n): Array[T]
  - RDD<T>#takeSample(n): Array[T]
  - RDD<T>#takeOrdered(n): Array[T]

Running a complex DAG

```
PairFunction<String, String, String> lineParser =
    (PairFunction<String, String, String>) line -> {
    String[] parts = line.split(",");
    return new Tuple2<>(parts[0], parts[1]);
};
JavaPairRDD<String, Iterable<String>> input1 =
sc.textFile("file1")
    .mapToPair(lineParser)
    .groupByKey();

JavaPairRDD<String, String> input2 = sc.textFile("file2")
    .mapToPair(lineParser)
    .filter(record -> record._1.equals("200"));

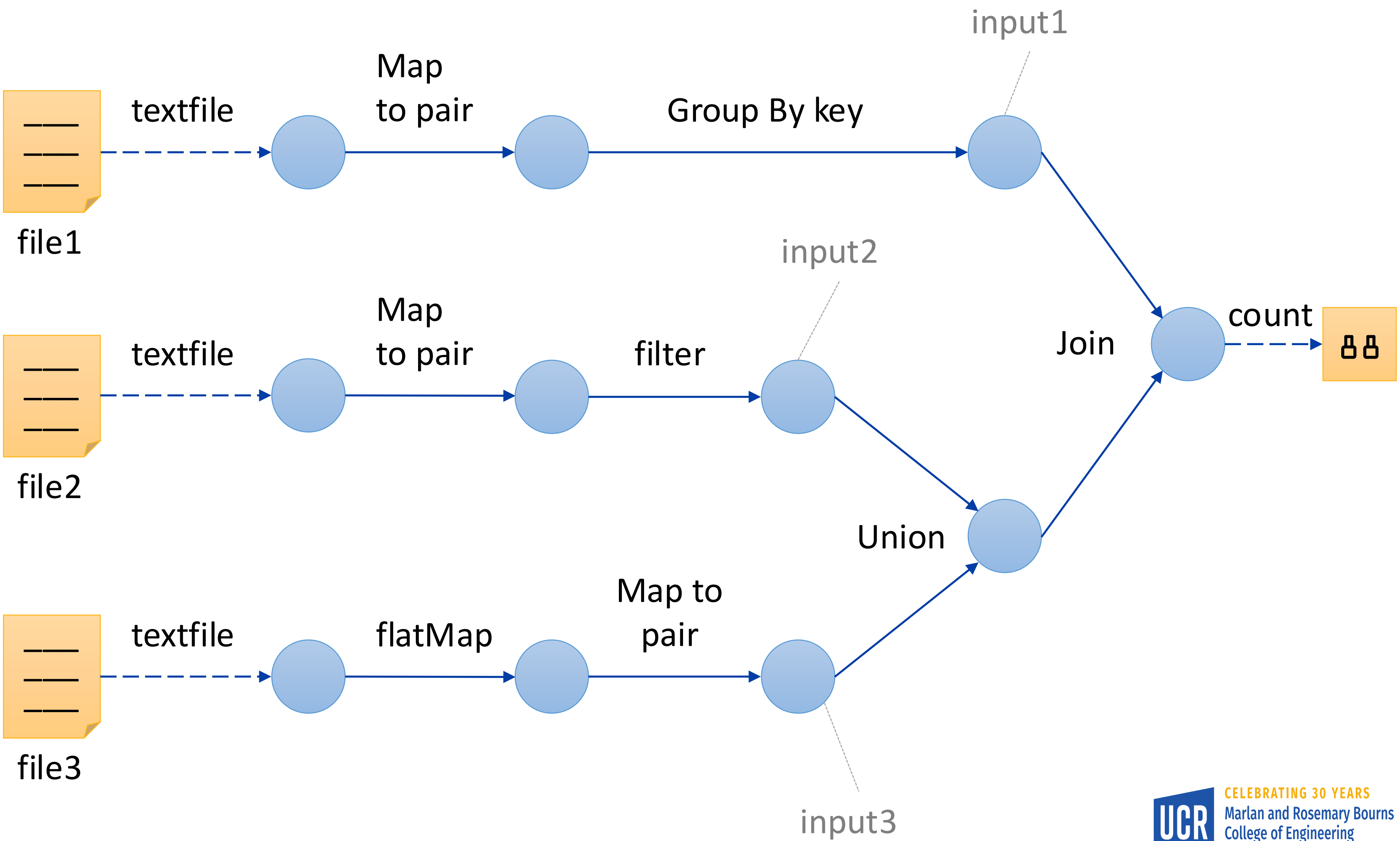
JavaPairRDD<String, String> input3 = sc.textFile("file3")
    .flatMap(line -> Arrays.asList(line.split(";")).iterator())
    .mapToPair(lineParser);

long count = input2.union(input3).join(input1).count();
```

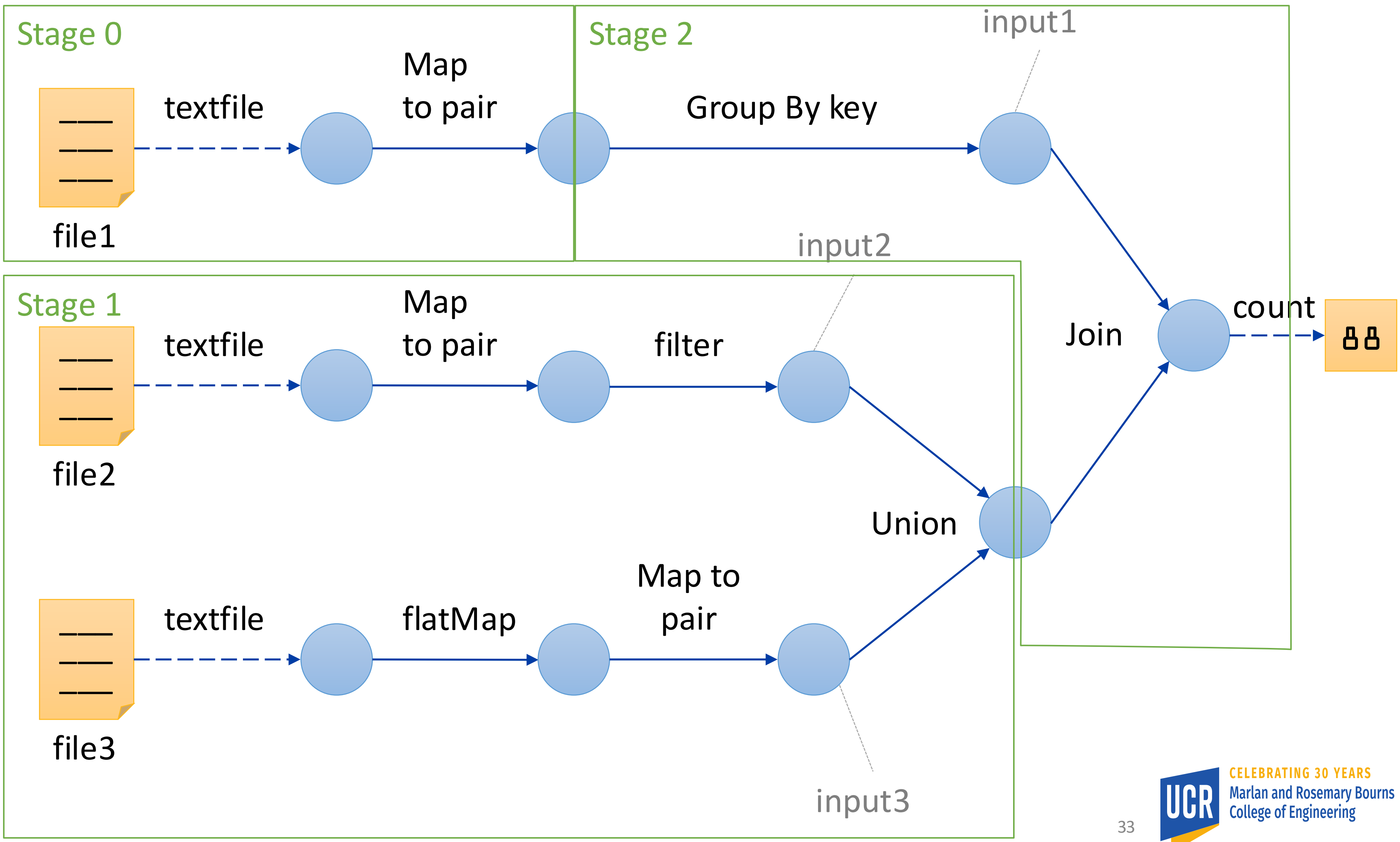
Practice Questions

- Convert the short program provided earlier into a DAG.
- Break down the DAG into stages. How many stages will this program have?

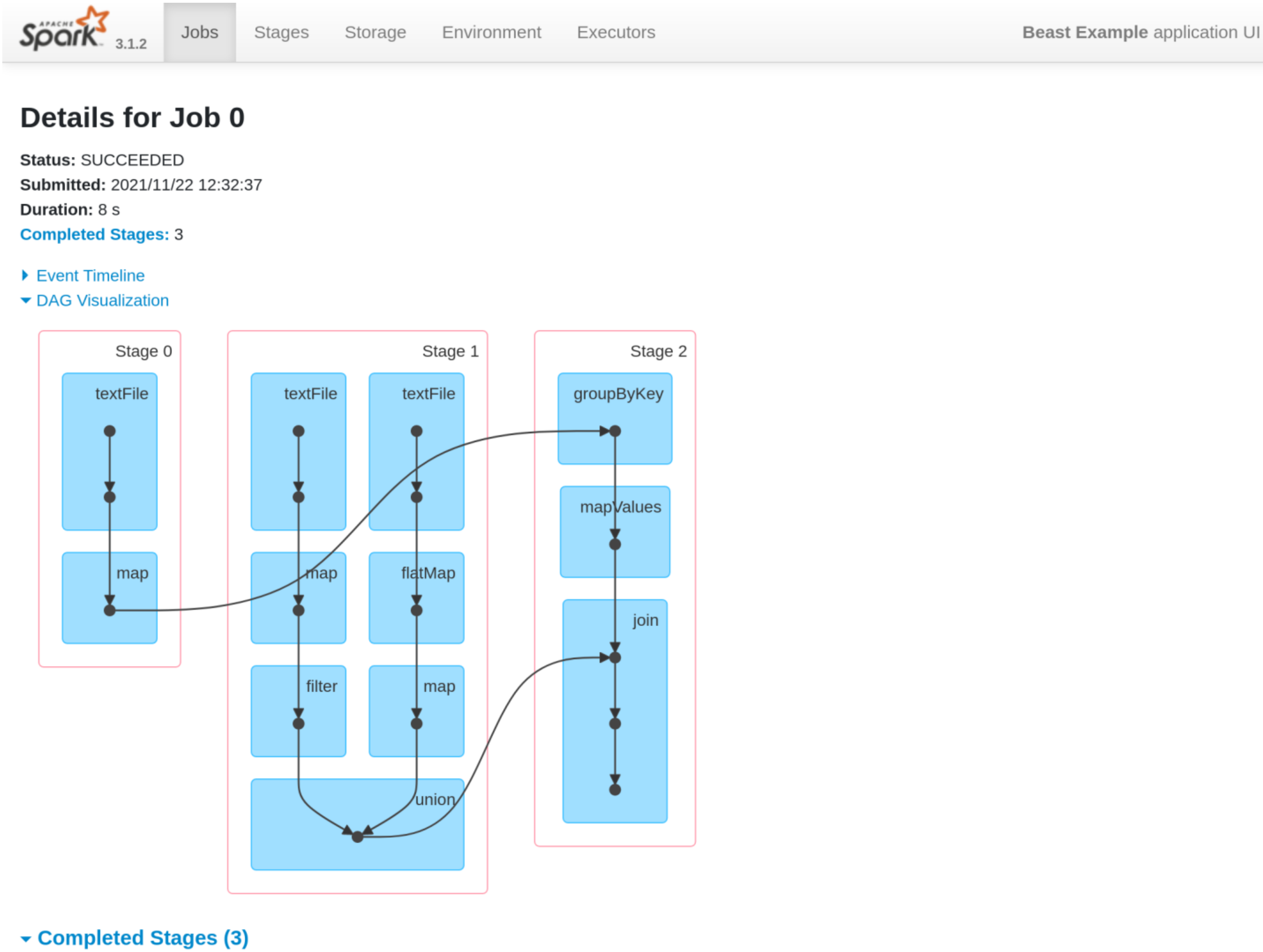
Answer - Overall DAG



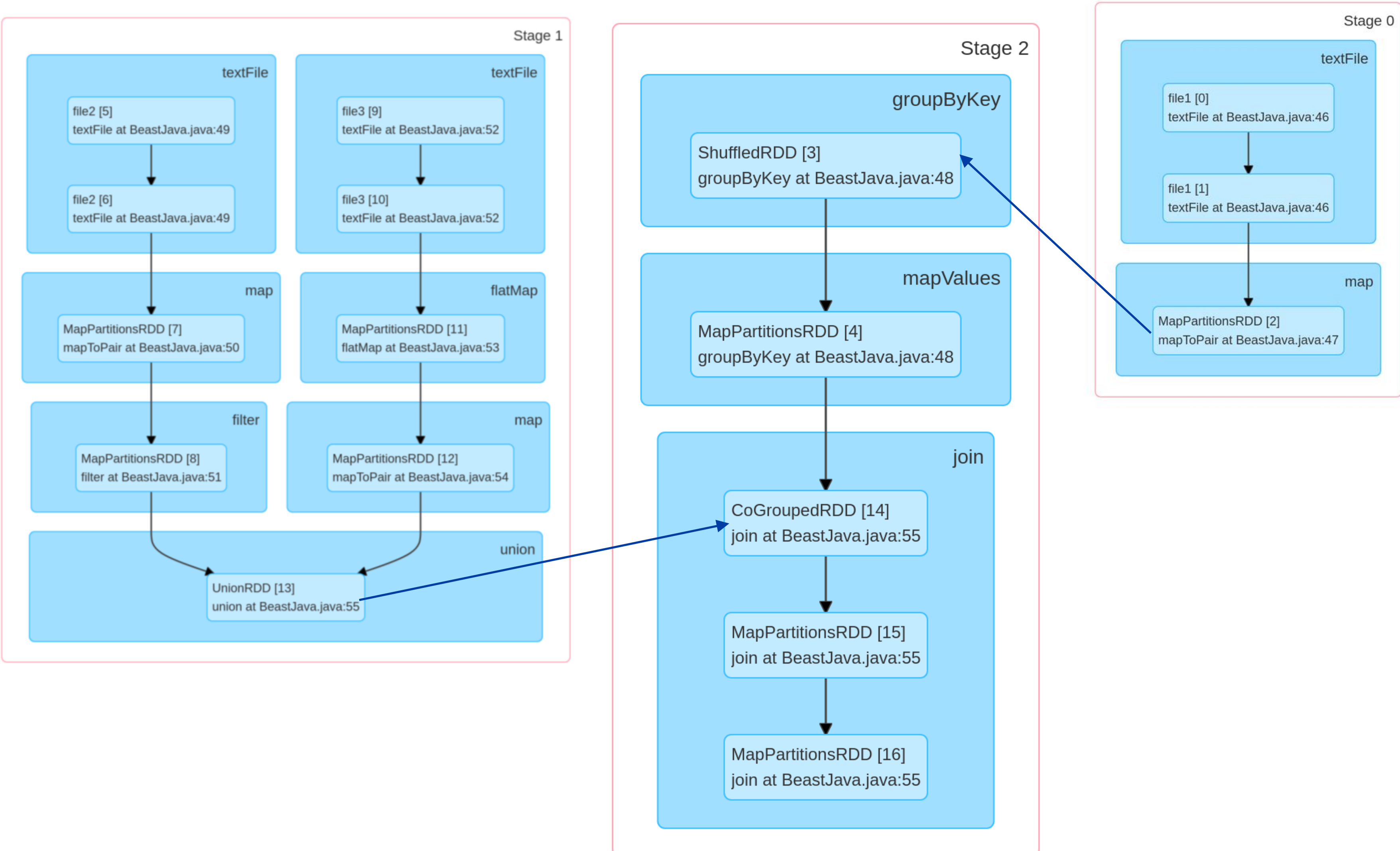
Answer - Overall DAG



Answer – Overall DAG



Answer – Stages



CELEBRATING 30 YEARS
Marlan and Rosemary Bourns College of Engineering

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