Concepts and Models of Parallel and Data-centric Programming

Apache Spark – Resilient Distributed Datasets

Lecture, Summer 2020

Simon Schwitanski
Dr. Christian Terboven

Outline

Resilient Distributed Datasets (RDDs)

- Immutable (read-only) collection of objects
- Partitioned across set of machines
- Abstraction of distributed memory
- Represented by a `JavaRDD` object with corresponding type
- Created through deterministic operations on storage data or existing RDD
- Construction in four ways
  1. Read a file from shared file system (e.g., HDFS)

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
```

Resilient Distributed Datasets (RDDs) (2)

2. "Parallelizing" a Java array resp. list in the driver program

```
Integer[] data = {1, 2, 3, 4, 5, 6};
JavaRDD<Integer> dataRDD = sc.parallelize(Arrays.asList(data));
```

3. Transforming an existing RDD

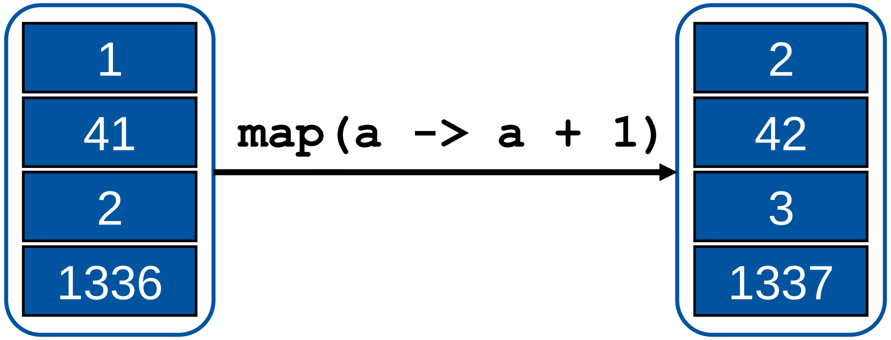
```
JavaRDD<Integer> newRDD = myRDD.map(a -> a + 1);
```

4. Changing the persistence of an existing RDD (*persist* or *cache* action)

```
JavaRDD<Integer> cachedRDD = dataRDD.persist();
```

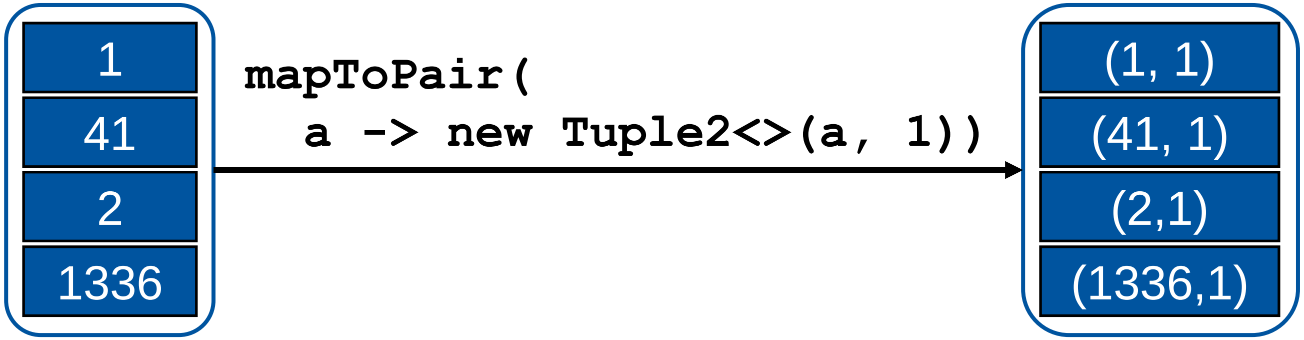
RDD Transformations (1)

- RDD transformation: Dataset with elements of type A transformed into a dataset with elements of type B
- *map(func)* : Apply *func* to each element of the RDD (one-to-one)

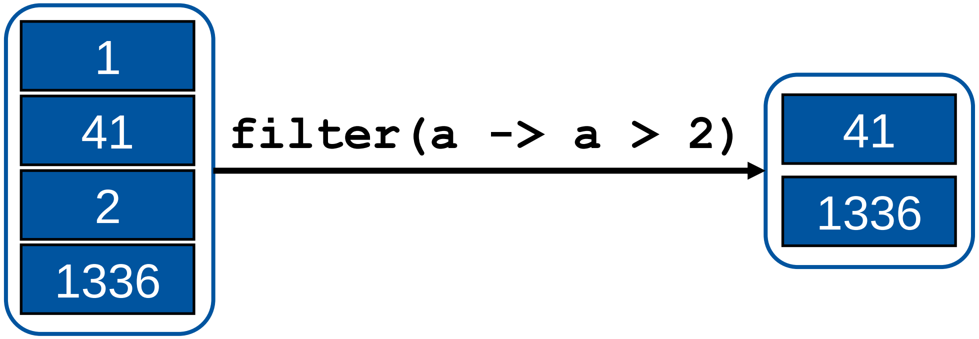


RDD Transformations (2)

- *mapToPair(func)* : Like *map(func)*, but maps to (K,V) pairs

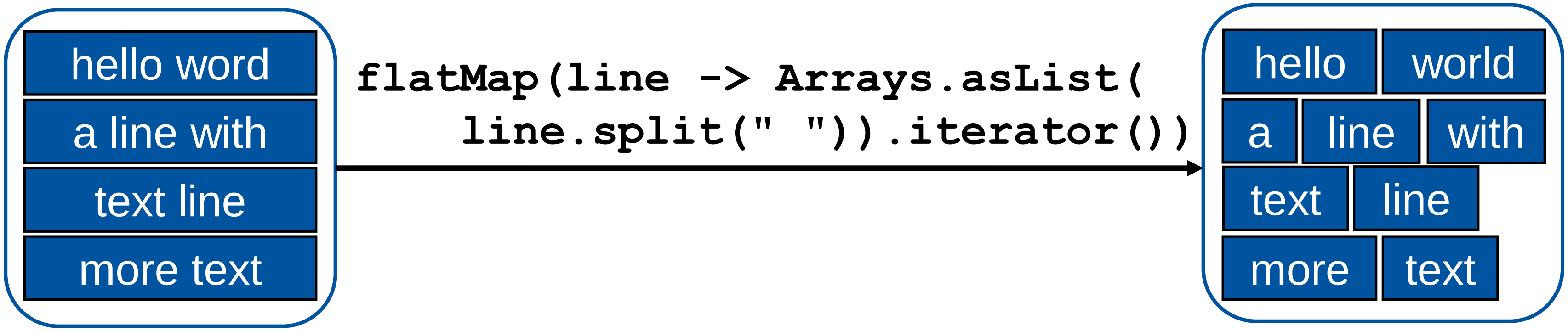


- *filter(func)* : Return those elements for which *func* gets true

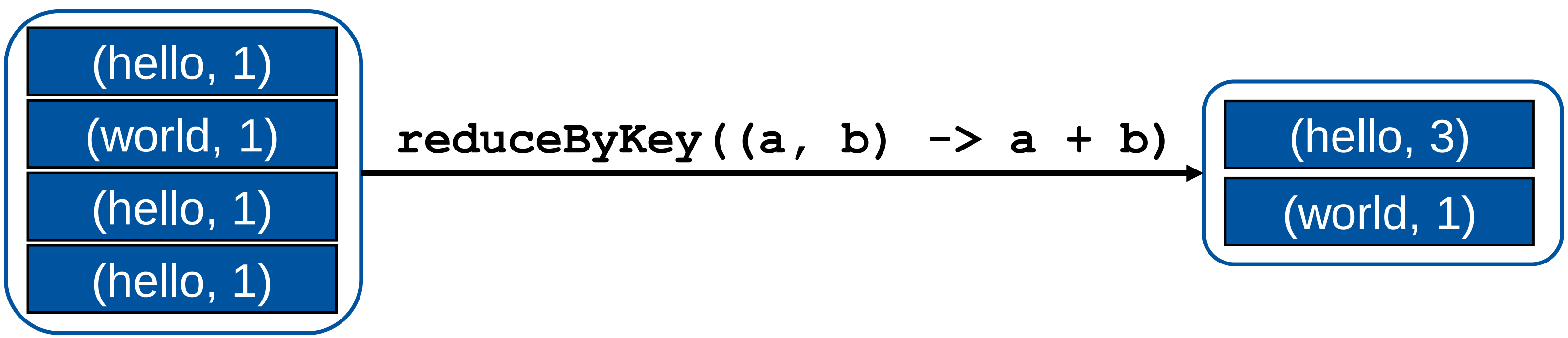


RDD Transformations (3)

- *flatMap(func)* : Similar to *map*, but each input can be mapped to zero or more outputs (one-to-many)

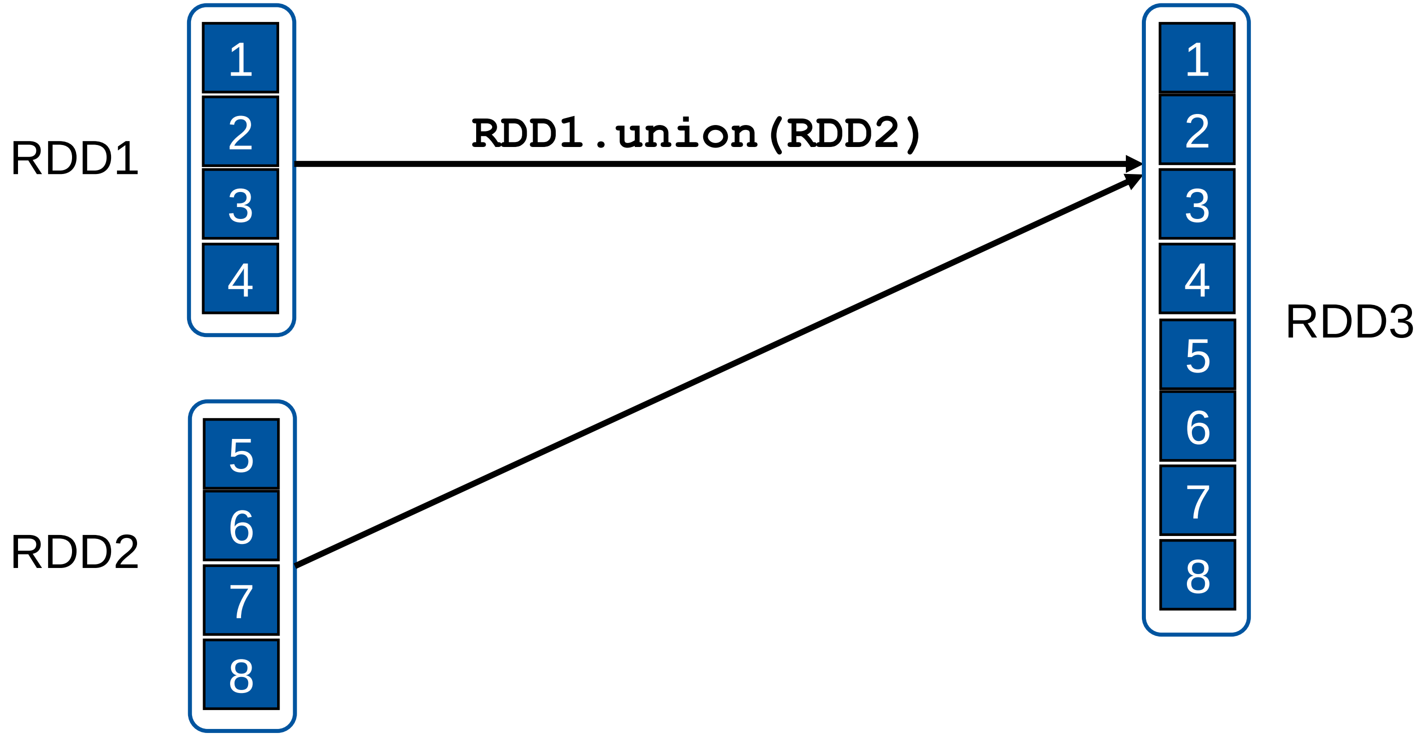


- *reduceByKey(func)* : Called on dataset of (K, V) pairs, returns dataset of (K, V) pairs where values for each key are aggregated using reduce function *func*



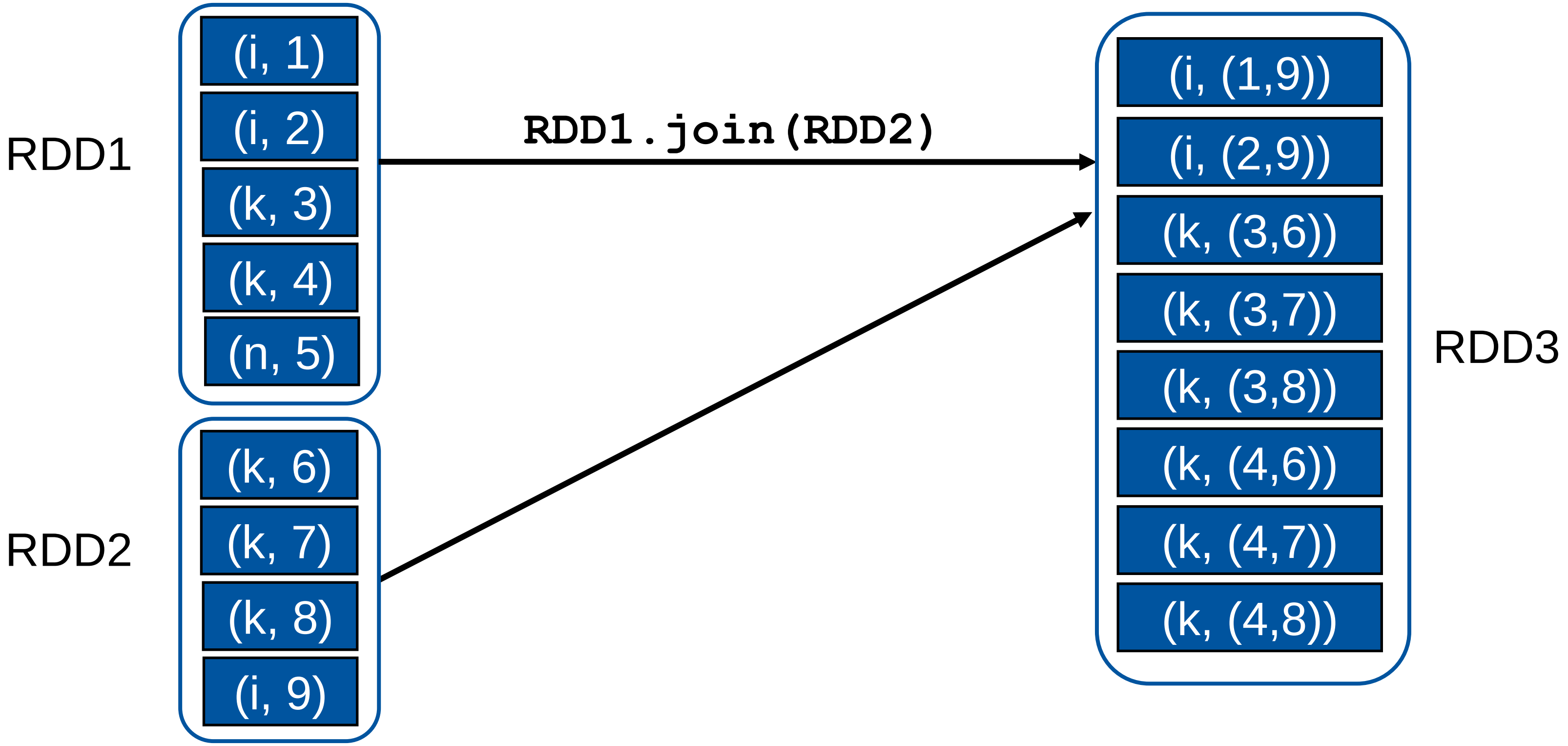
RDD Transformations (4)

- *union(otherDataset)* : Return new dataset containing union of the elements in the source dataset and the argument



RDD Transformations (5)

- *join(otherDataset)* : Called on dataset of (K, V) pairs, returns dataset of (K, (V, W)) pairs with all pairs of elements for each key (inner join)
- *leftOuterJoin*, *rightOuterJoin*, *fullOuterJoin* analogous

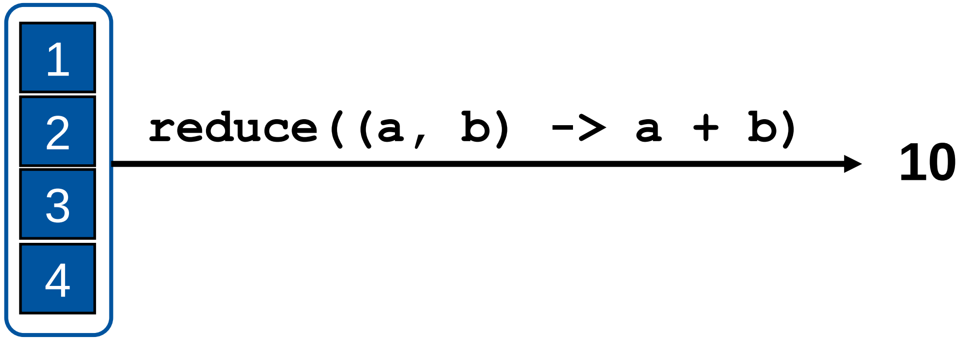


RDD Transformations (6)

- Full list of transformations: https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/JavaRDD.html
- **Important:** RDD transformations are **lazy operations**
  - RDD transformations are not computed ("materialized") at all time
  - Instead: RDD stores information about how it is derived in a so called *lineage*
  - Each RDD can compute its partitions via its *lineage graph* from the original data

RDD Actions

- RDD action: Launch a computation to return a value to the program or write data to storage
  - *collect()* : Return all elements of the RDD as array to the driver program
  - *count()* : Return the number of elements in the RDD
  - *reduce(func)* : Aggregate elements of the RDD using reduce function *func*
  - *save(path)* : Write elements of RDD to given location (local file systems, HDFS, etc.)
  - …
- Actions *always* lead to an actual computation (materialization)



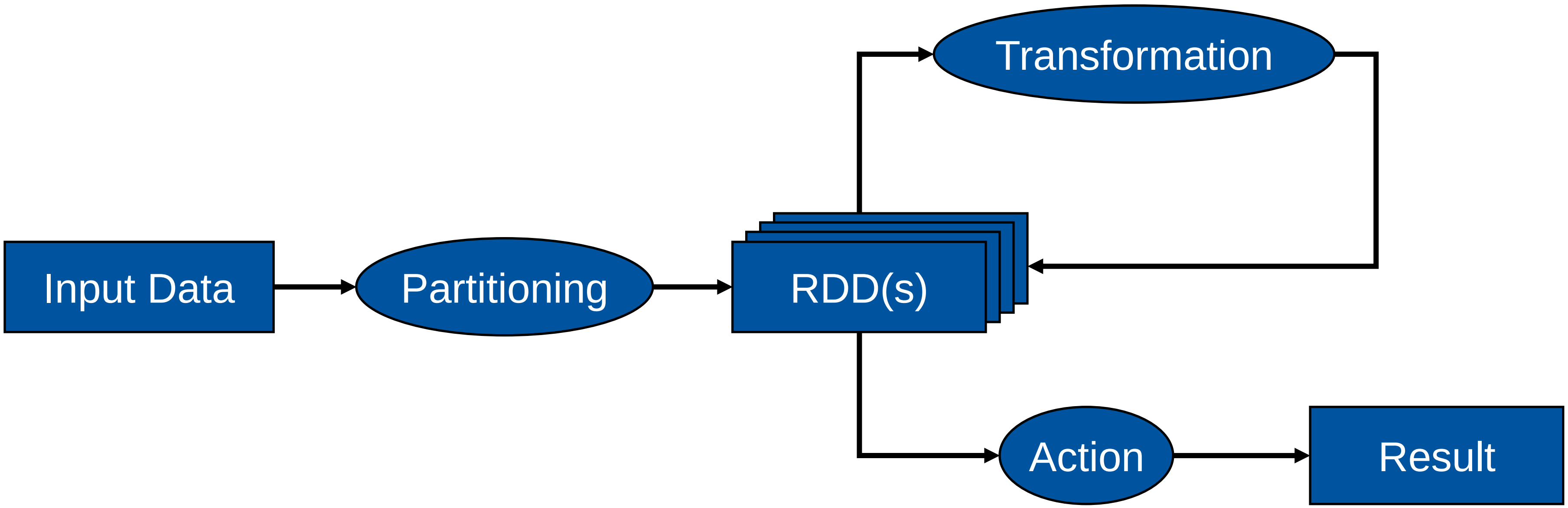
RDD Function Signatures

Transformations	$map(f: T \Rightarrow U)$	$RDD[T] \Rightarrow RDD[U]$
	$mapToPair(f: T \Rightarrow (K, V))$	$RDD[T] \Rightarrow RDD[(K, V)]$
	$filter(f: T \Rightarrow Bool)$	$RDD[T] \Rightarrow RDD[T]$
	$flatMap(f: T \Rightarrow Seq[U])$	$RDD[T] \Rightarrow RDD[U]$
	$groupByKey()$	$RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$
	$reduceByKey(f: (V, V) \Rightarrow V)$	$RDD[(K, V)] \Rightarrow RDD[(K, V)]$
	$union()$	$(RDD[T], RDD[T]) \Rightarrow RDD[(K, V)]$
	$join()$	$(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W)]$
Actions	$count()$	$RDD[T] \Rightarrow Long$
	$collect()$	$RDD[T] \Rightarrow Seq[T]$
	$reduce(f: (T, T) \Rightarrow T)$	$RDD[T] \Rightarrow T$
	$save(path: String)$	Outputs RDD to a storage system

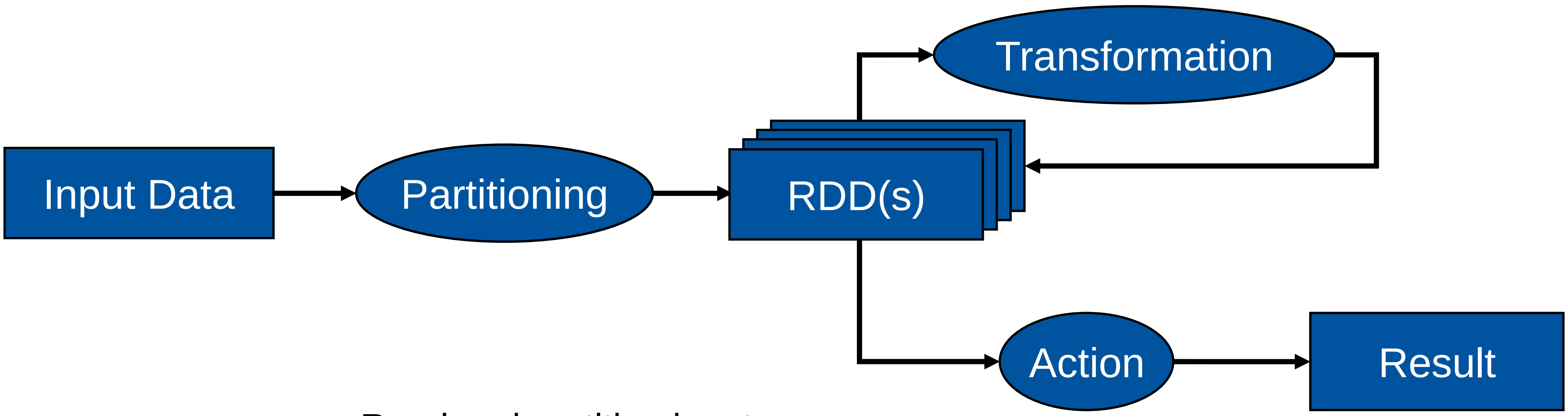
Note: $Seq[T]$ denotes sequence of elements of type T

Spark Program – Typical Workflow

1. Create RDDs from external data (e.g. HDFS files).
2. Transform the RDDs with the desired operations.
3. Perform RDD action to output the RDD(s) for external data sources.



Word Count Example – Revisited



Read and partition input data to create RDD

Transformations

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```

Action

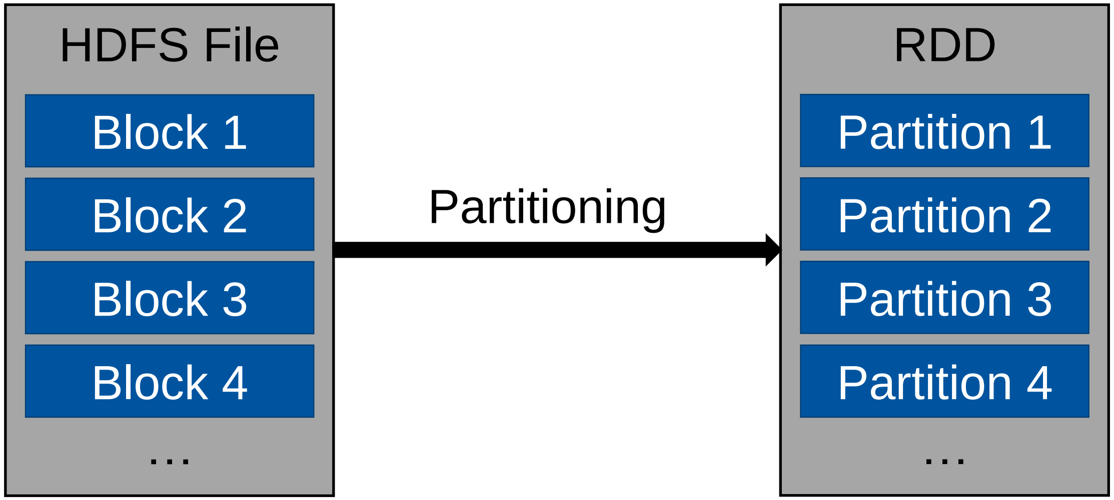
Store result in file

RDD Representation

- Choose a representation for RDDs which can track the *lineage* between transformations
  - From which other RDDs is an RDD derived using which transformation?
  - Important for efficiency and fault tolerance
- RDD contains four pieces of information
  1. Set of partitions (atomic pieces of the dataset)
  2. Set of dependencies on parent RDDs (through transformations)
  3. Function for computing the dataset based on its parents
  4. Metadata about partitioning scheme and data placement
- Dependencies between RDDs form the *lineage graph*

RDD Representation – Partitions (1)

- Partitions: Atomic pieces of RDD, potentially stored on different machines
- Typically: Input data stored on distributed file system (e.g., HDFS)
- RDD representing an HDFS file: One partition for each HDFS block



RDD Representation – Partitions (2)

- Partitioning of data can also be controlled by user (as in MapReduce)
  - User can define custom partitioner
- Two kinds of predefined partitioners
  - Hash-based: Use *hashCode()* method
  - Range-based: Partition (sortable) elements in equal ranges
- Default partitioner: Hash-based
- (Re-)partitioning of an RDD by calling action *repartition()*
- RDDs with same partitioner are called *co-partitioned*.

RDD Representation – Lineage Graph

- Lineage graph: Directed Acyclic Graph (DAG)
  - Vertices: RDD objects
  - Edges: RDD transformations (dependencies of RDDs)
- Represents the “history” of an RDD

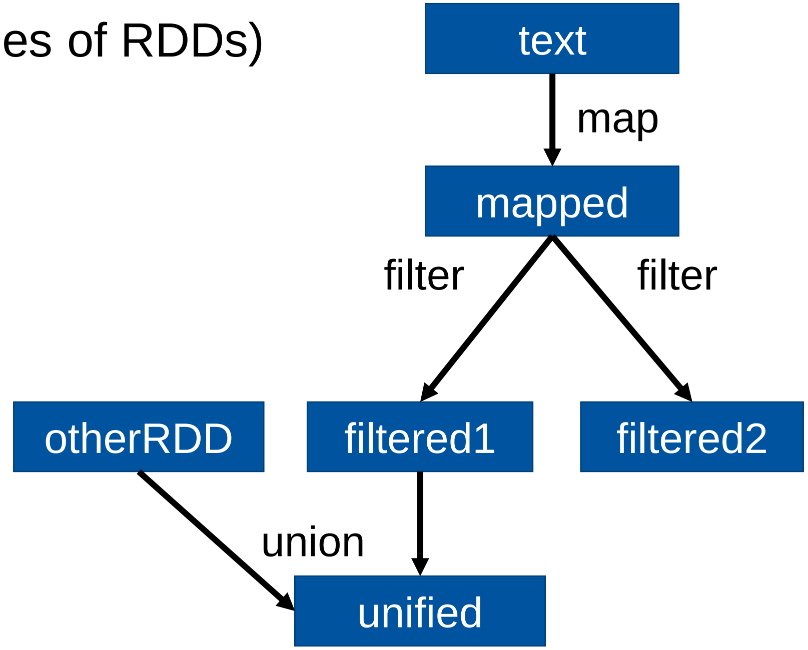
```
JavaRDD<String> text = sc.textFile(...);

JavaRDD<String> mapped = text.map(...);

JavaRDD<String> filtered1 =
   mapped.filter(...);

JavaRDD<String> filtered2 =
   mapped.filter(...);

JavaRDD<String> unified =
   filtered1.union(otherRDD);
```



RDD Representation – Dependencies (1)

- Differentiate *narrow* and *wide dependencies* between RDDs
- Narrow dependency
  - Each partition of parent RDD is used by **at most one** partition of child RDD
  - Example transformations: *map*, *filter*, *union*
- Transformations with narrow dependencies can be pipelined on one node

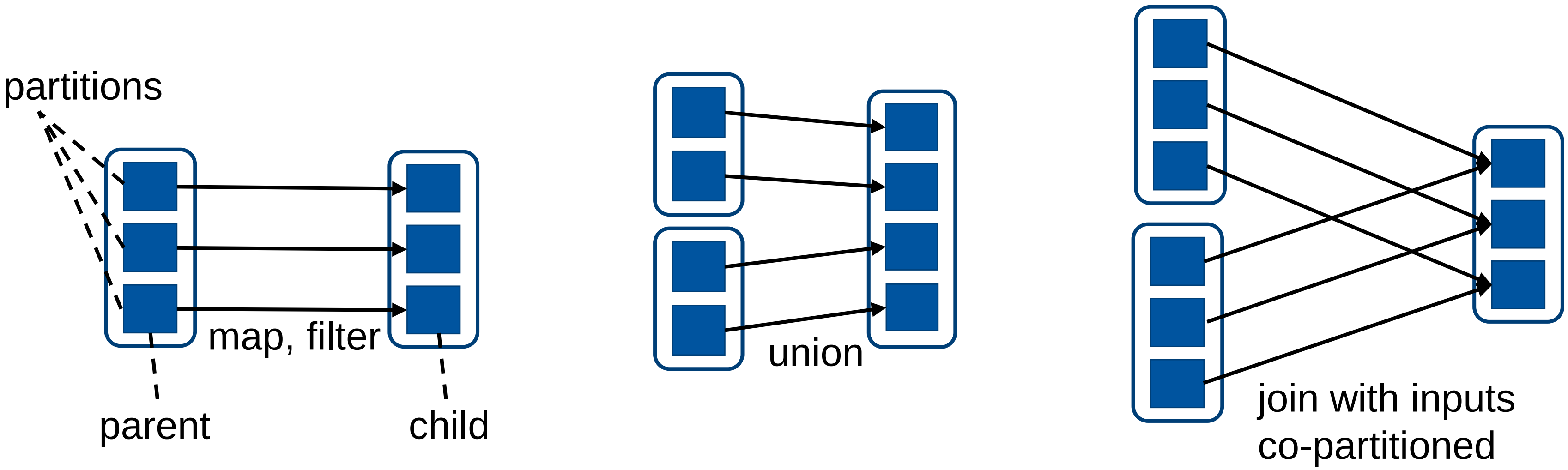
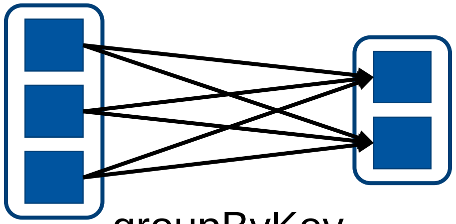


Image Source: Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." NSDI2012: 15-28

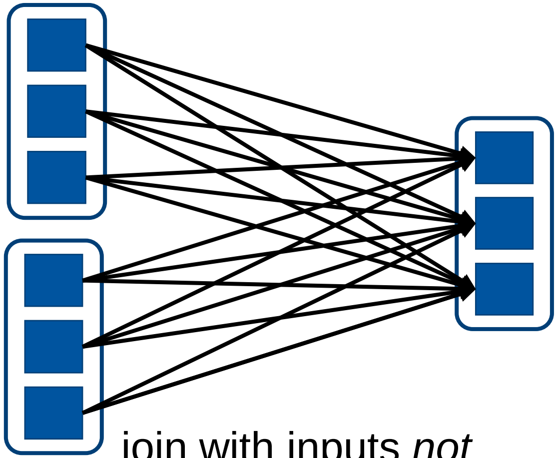
RDD Representation – Dependencies (2)

- Wide dependency
  - Multiple child partitions may depend on one partition of the parent RDD
  - Example transformations (which *typically* have wide dependencies, but not in every case): *reduceByKey*, *groupByKey*
- Wide dependencies indicate that shuffling across nodes is required (high network I/O required, as for MapReduce shuffling)

Assumption: Input not already grouped and not partitioned with a partitioner.



groupByKey



join with inputs *not* co-partitioned

Image Source: Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." NSDI2012: 15-28

RDD Representation – Dependencies (3)

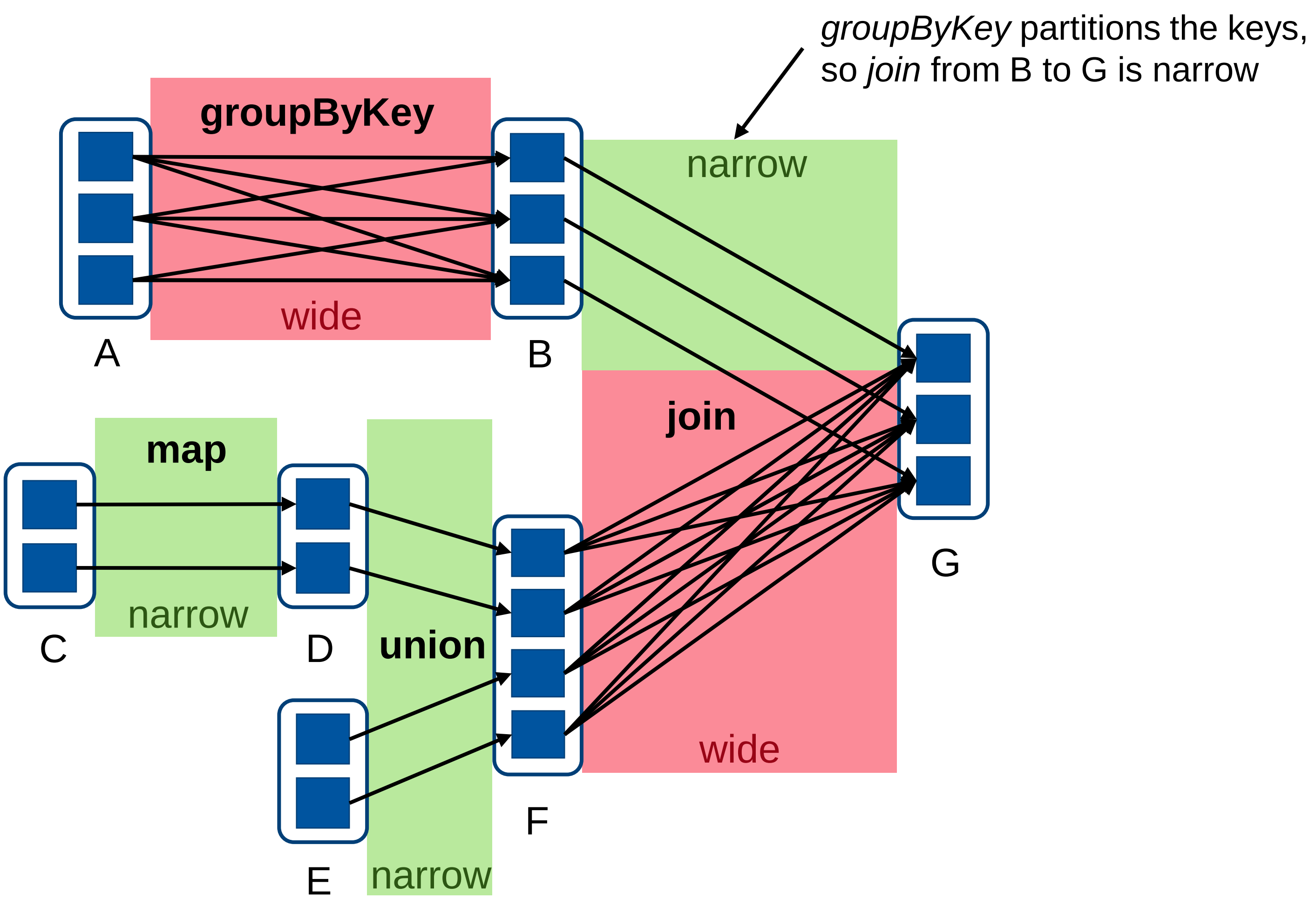


Image Source: https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies

RDD Persistence

- Iterative algorithms benefit from persisting (caching) RDDs in memory
- Use *persist()* method to change persistence of RDD
  - If RDD is materialized: Workers will keep dataset in memory
- If not enough RAM available: Spill (part of) dataset to disk
- Example: Logistic regression (iterative classification algorithm) in Scala

```
// Read points from a text file and cache them
val points = spark.textFile(...).map(parsePoint).persist()
// Initialize w to random D-dimensional vector
var w = // Random initial vector
// Run multiple iterations to update w
for (i <- 1 to ITERATIONS) {
    val gradient = points.map{ p =>
      p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
    }.reduce((a,b) => a+b)
    w -= gradient
}
```

Code Source: Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.” NSDI2012: 15-28

Logistic Regression Performance in Hadoop and Spark

- Logistic regression job: 29 GB on 20 “m1.xlarge” AWS EC2 nodes (4 cores each)
- Hadoop: Each iteration 127s
- Spark: First iteration 174s (Scala overhead), subsequent iterations 6s
- Huge performance benefit due to RDD persistence

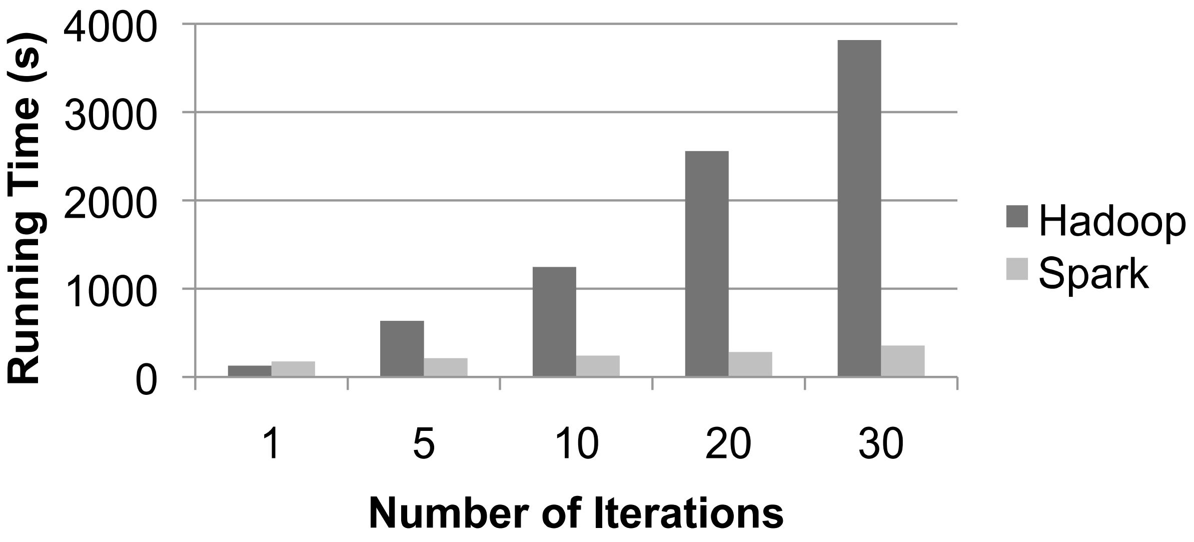


Image Source: Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. “Spark: Cluster Computing with Working Sets”. HotCloud 2010