Apache Spark

CS 6240 - Parallel Data Processing - Spring 2017

Presented by: Joseph Sackett
jsackett@ccs.neu.edu

Definition

Apache Spark™ is a fast and general engine for large-scale data processing. [1]

- Fast
  - Compared to what?
  - How?
    - In-memory - RDDs
    - Lazy evaluation - optimizations
- General
  - Is this good?
  - Smaller, numerous, primitive operations
  - Allows vertical specializations (e.g., MLlib, GraphX)

Framework API over Distributed Computation Runtime

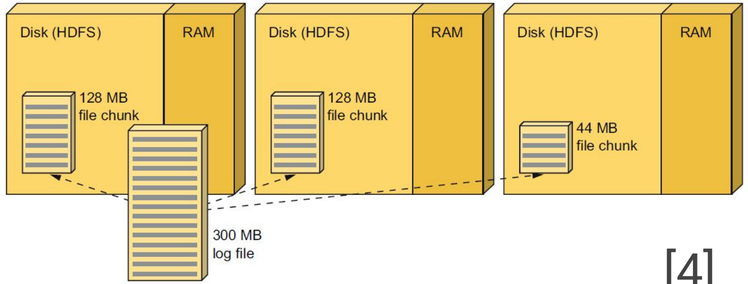
Background [2]

- Supports larger scale data processing workloads
- Keeping the scalability and fault tolerance of previous systems
- Enables streaming and interactive applications, while specialized systems only support simple one-pass computations (e.g., aggregation or SQL queries)
- Allows computations to be combined
- MapReduce poorly suited some workloads, leading to specialized models:
    - F1 - scalable distributed RDBMS providing consistency and usability of SQL databases
    - Impala - scalable, distributed SQL query engine over HDFS & HBase data
    - Pregel - bulk-synchronous parallel (BSP) model for iterative graph algorithms
    - GraphLab - parallel programming abstraction for graph & ML algorithms
    - Storm, MillWheel - scalable, fault tolerant stream processing
    - Piccolo - prog. model for distributed, in-memory, key-value data

Resilient Distributed Datasets

What are RDDs ? [3]

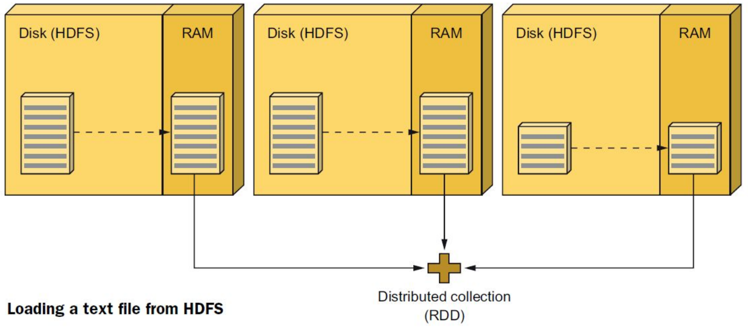
- Immutable distributed collection of objects
- Data split into multiple partitions to reside and have computations applied at different nodes
- Contain any type of objects
- Created by:
  - loading an external dataset
  - distributing a collection of objects
  - transforming existing RDD
- Remembers the graph of operations used to build it:
  - avoids data replication
  - can recompute parts of RDD on node failure



[4]

Storing a 300 MB log file in a three-node Hadoop cluster

```
val lines = sc.textFile("hdfs://path/to/the/file")
```



Loading a text file from HDFS

Resilient Distributed Datasets (RDDs)

Why RDDs ?

- Clean programming abstraction - seamlessly integrates with Scala, et al.
- Extension to MR; adds efficient data sharing across parallel computation stages
- Supports batch, iterative and streaming computations in same runtime
- Up to 100x speedup over MR due to lazy optimizations and in-memory residence
- Stronger and extended fault tolerance model

Scala Spark Program - Initialization

```
// Initialize, connect to cluster URL ("local" - standalone)

val conf = new SparkConf().setMaster("local").setAppName("My App")

val sc = new SparkContext(conf)
```

Function Literals (i.e., lambdas)

Scheme: `(lambda (x y) (+ x y))    // what is type ??`

Scala: `(x, y) => x + y           // what is type ??`

Scala Spark Application - Temperature Average

```
val input = sc.textFile("input")
val maxTemps = input.map(line => line.split(","))
  .filter(fields => fields(2) == "TMAX").persist
val numTemps = maxTemps.count
val sumTemps = maxTemps.map(fields => Integer.parseInt(fields(3)))
  .reduce((sum, temp) => sum + temp)
println("Avg Temp: " + sumTemps / numTemps)
```

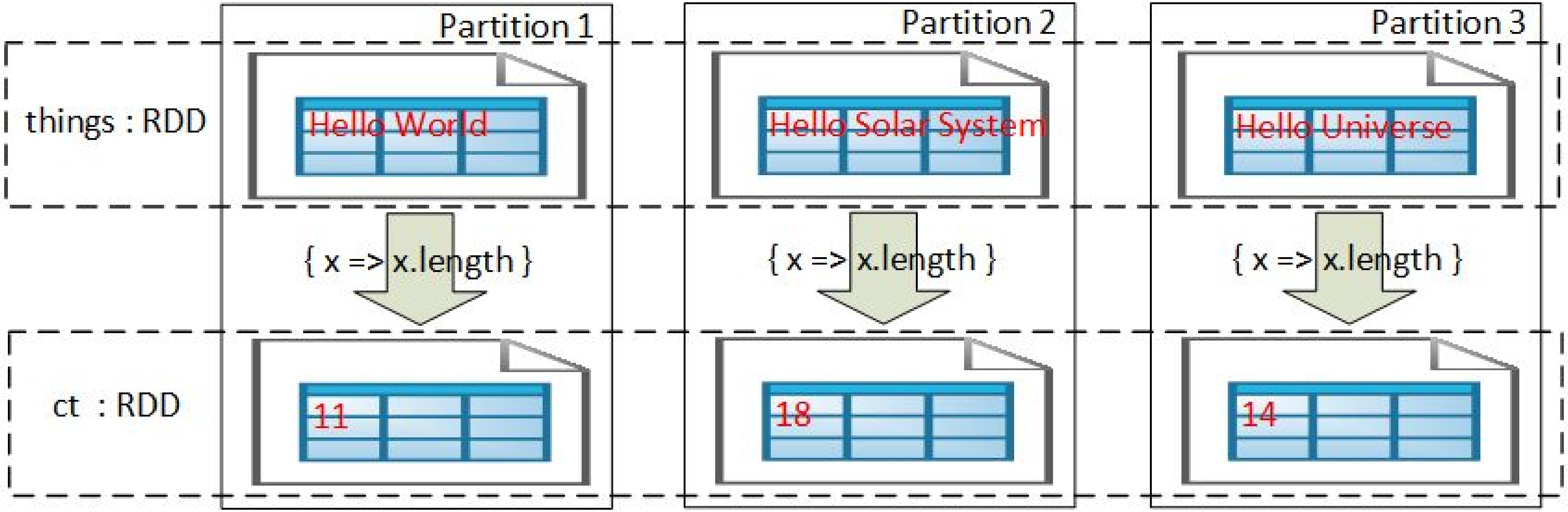
Transformations & Actions

RDDs offer two types of operations:

1. **Transformations** construct a new RDD from a previous one. Spark uses lazy evaluation, encoding RDDs as graph of operations until used in an action. Spark optimizes the graph of transformations to efficiently compute the data.
2. **Actions** compute and collect a result to the driver program or storage. They force evaluation of a lazy RDD.

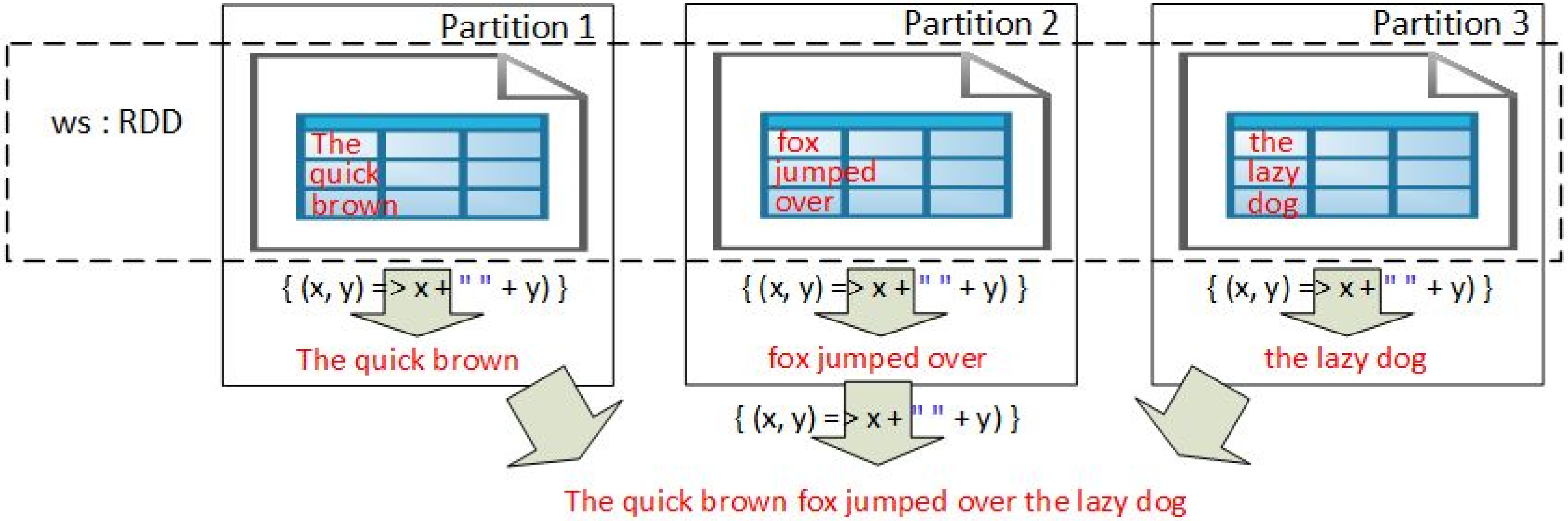
RDDs are recomputed for each action. RDDs used in multiple actions can be persisted in memory for reuse in future actions, via RDD.persist()
Persisting RDDs on disk is also possible.

Map Example



```
val things = sc.parallelize(Array("Hello World","Hello Solar System","Hello Universe"),3)
val ct = things.map(x => x.length) // map func. to RDD, f type?
ct.collect.foreach(println)        // action forces RDD
ct.saveAsTextFile("output")        // action forces RDD again
```

Reduce Example



```
val ws = sc.parallelize(Array("The", "quick", "brown", "fox", "jumped",
                        "over", "the", "lazy", "dog"), 3)
val s = ws.reduce((x, y) => x + " " + y)
println(s)
// how to reduce prior example ??
```

Passing Functions

- Transformations and actions often take a function as an argument.
- Functional operations parallelize easily across cluster.
- In Scala, we can pass in functions defined:
  - inline function literal
  - method references
  - static functions
- Scala Closures
  - function and referenced data (i.e., environment) must be serializable
  - passing an object method includes reference to *this*
  - define fields as local variables to avoid the object in the environment

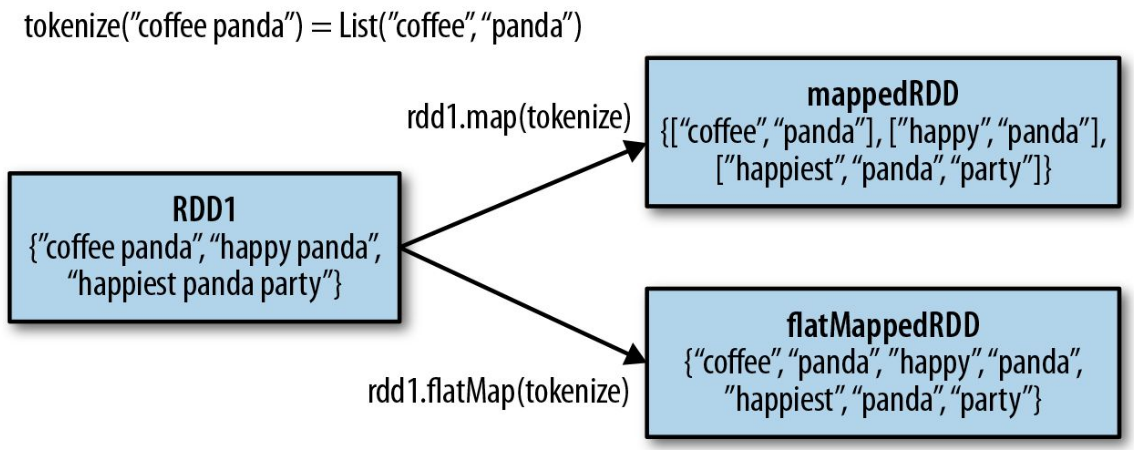
Closures

[3]

```
class SearchFunctions(val query: String) {
    def isMatch(s: String): Boolean = s.contains(query)

    def getMatchesFunctionReference(rdd: RDD[String]): RDD[String] = {
        // Problem: "isMatch" means "this.isMatch", so we pass all of "this"
        rdd.map(isMatch)
    }

    def getMatchesFieldReference(rdd: RDD[String]): RDD[String] = {
        // Problem: "query" means "this.query", so we pass all of "this"
        rdd.map(x => x.split(query))
    }

    def getMatchesNoReference(rdd: RDD[String]): RDD[String] = {
        // Safe: extract just the field we need into a local variable
        val queryRef = this.query
        rdd.map(x => x.split(queryRef))
    }
}
```

Simple Transformations

- **filter** RDD[A].filter(f : A => Boolean) : RDD[A]
- **map** RDD[A].map(f : A => B) : RDD[B]
- **flatMap** RDD[A].flatMap(f : A => Seq[B]) : RDD[B]



tokenize("coffee panda") = List("coffee", "panda")
rdd1.map(tokenize)
mappedRDD
{["coffee", "panda"], ["happy", "panda"], ["happiest", "panda", "party"]}
RDD1
{"coffee panda", "happy panda", "happiest panda party"}
rdd1.flatMap(tokenize)
flatMappedRDD
{"coffee", "panda", "happy", "panda", "happiest", "panda", "party"}

[3]

- **mapPartitions** RDD[A].map(f : A => B) : RDD[B]

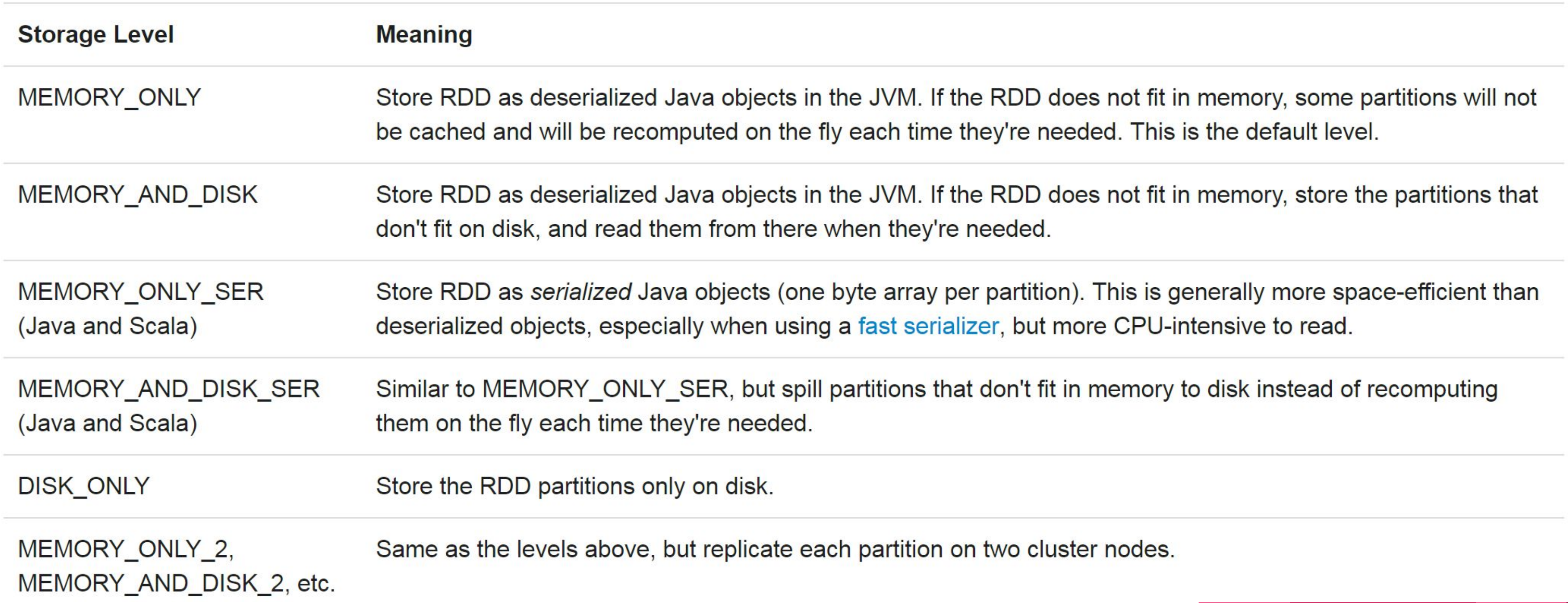
Simple Actions

- **count** - element count RDD[_].count : Integer
- **first** - first element RDD[A].first : A
- **collect** - all elements RDD[A].collect : Array[A]
- **take** - first n elements RDD[A].take(n : Integer) : Array[A]
- **takeOrdered** - first n elements, custom ordering
  RDD[A].takeOrdered (n : Integer)(f : Ordering[A]) : Array[A]
- **reduce** - accumulates to single value, same type as held in RDD
  RDD[A].reduce(f : (A, A) => A) : A
- **aggregate** - accumulates to single value, arbitrary type
  RDD[A].aggregate(s : B)(seq : (A, B) => B)(comb : (B, B) => B) : B

Scala Spark App - Temp Average Aggregate

```
val sumCount = sc.textFile("input")
  .map(line => line.split(","))
  .filter(fields => fields(2) == "TMAX")
  .map(fields => Integer.parseInt(fields(3)))
  .aggregate((0, 0))(    // aggregate can reduce with multiple accumulators
            (acc, value) => (acc._1 + value, acc._2 + 1),
            (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))

println("Avg Temp: " + sumCount._1 / sumCount._2.toDouble)
```

RDD.persist(storage_level)



Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.

RDD.cache = RDD.persist(MEMORY_ONLY)

Key-Value Pairs

- Familiar format:
  - Databases
  - Maps
- Data type required for PairRDDs, allowing:
  - *ByKey aggregation operations (e.g., sum, count).
  - join operations
  - parallel operations by key
- Better partitioning of PairRDD by key to allow related data to be localized to reduce communication costs.

PairRDDs

PairRDD contains key/value pairs.

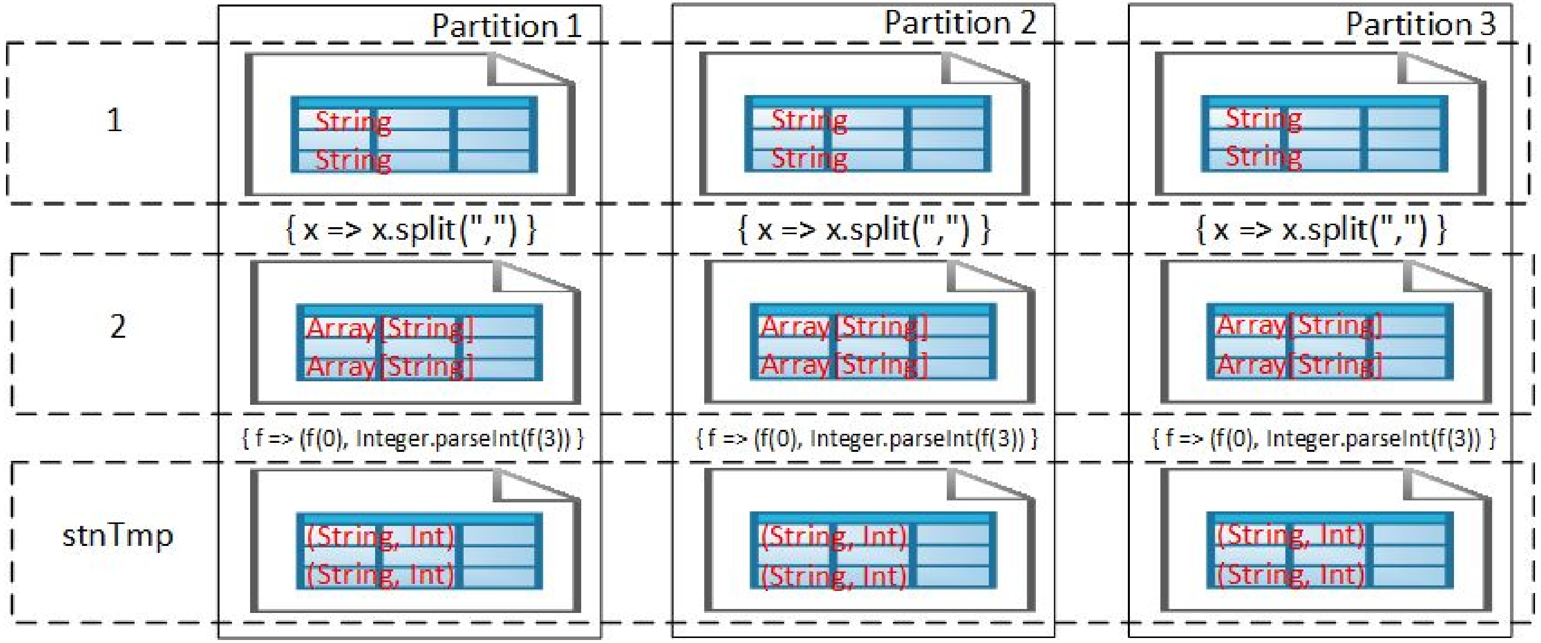
This is represented as a tuple in Scala; arbitrary types but key must be hashable:

```
val kv = (“key”, “value”)
```

Example, creating a key value of temperature data:

```
ITE00100554,18001228,TMAX,35,,,E,

val stationTemp = (fields(0), Integer.parseInt(fields(3)))
```

Creating PairRDD



```
val stnTmp = sc.textFile("input", 3) // load data
  .map(line => line.split(","))      // tokenize
  .map(fields => (fields(0), Integer.parseInt(fields(3))))
```

Common PairRDD Operations

- **reduceByKey(func)** - Reduce values with the same key.
  PairRDD[(K, A)].reduceByKey(f: (x : A, y : A) => A) : PairRDD[(K, A)]
- **foldByKey()** - Merge values for keys using an assoc function and “zero value."
  PairRDD[(K, A)].foldByKey(s: A, f: (x : A, y : A) => A) : PairRDD[(K, A)]
- **combineByKey()** - combine elements for each key using custom aggregations.
  PairRDD[(K, A)].**combineByKey**(fCreate, fCombine, fMerge) : PairRDD[(K, B)]
- **mapValues(func)** - Apply a function to each value, returning any type.
  PairRDD[(K, A)].mapValues(f: A => B) : PairRDD[(K, B)]
- **flatMapValues(func)** - Apply a function returning an iterator to any type value, and for each element returned, produce a key/value entry.
  PairRDD[(K, A)].flatMapValues(f : A => Seq[B]) : PairRDD[(K, B)]
- Others: keys(), values(), sortByKey(), groupByKey()

PairRDD Set Operations

- **join** - Perform an inner join between two RDDs.
  rdd.join(other)
- **leftOuterJoin** - Perform a join between RDDs where result key must be present in pairRDD2.
  pairRDD1.leftOuterJoin(pairRDD2)
- **rightOuterJoin** - Perform a join between RDDs where result key must be present in pairRDD1.
  pairRDD1.rightOuterJoin(pairRDD2)
- **subtractByKey** - Remove elements with a key present in the other RDD.
  pairRDD1.subtractByKey(pairRDD2)
- **cogroup** - Group data from both RDDs sharing the same key.
  pairRDD1.cogroup(pairRDD2) : RDD[(K, (Iterable[V], Iterable[W]))].

Pair RDD Actions

- **countByKey**() - Count the number of elements for each key; returns Map.
- **collectAsMap**() - Collect the result as a Map of key/values.
- **lookup**(key) - Returns list of values associated with input key.

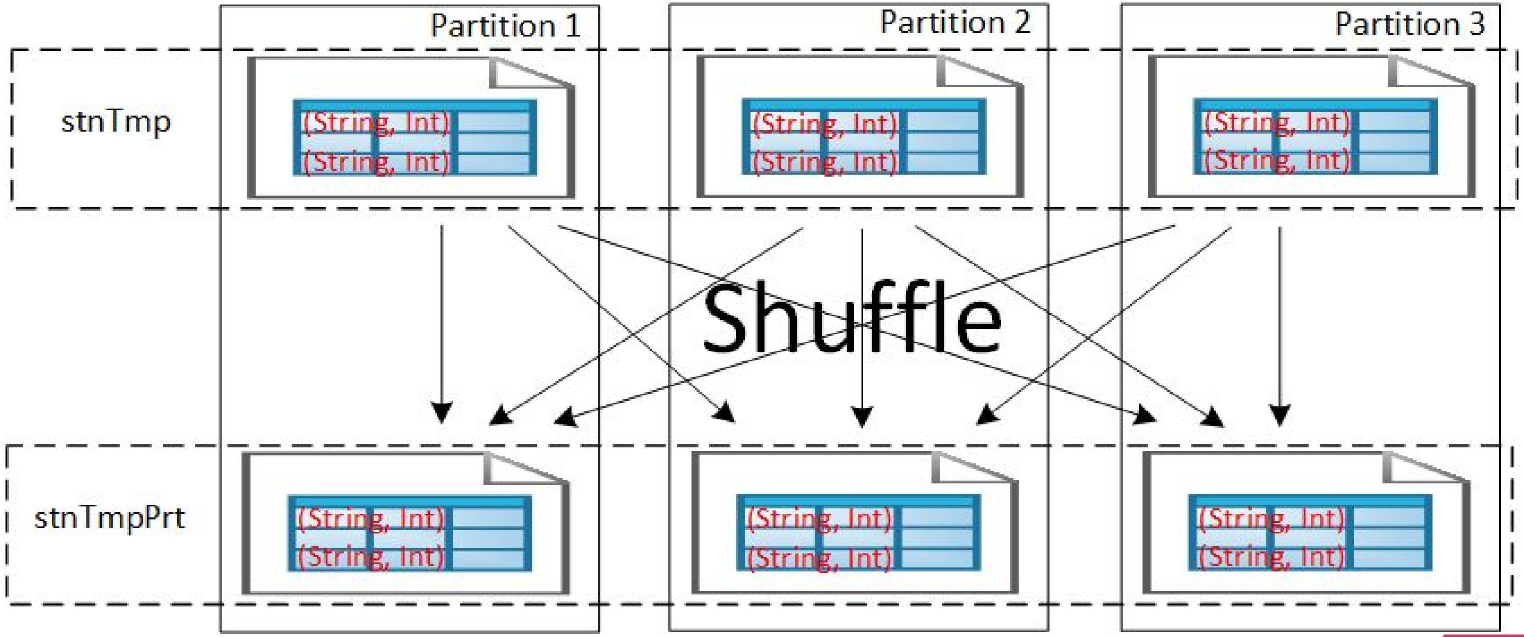
Scala Spark App - Temp Average By Key

```
val pairRDD = sc.textFile("input")
  .map(line => line.split(","))
  .filter(fields => fields(2) =="TMAX")
  .map(fields => (fields(0), Integer.parseInt(fields(3))))

val sumByKey = pairRDD.reduceByKey((sum, temp) => temp + sum)
val countByKey = pairRDD.foldByKey(0)((ct, temp) => ct + 1)
val sumCountByKey = sumByKey.join(countByKey)
val avgByKey = sumCountByKey.map(t => (t._1, t._2._1 / t._2._2.toDouble))

avgByKey.saveAsTextFile("output")
```

Data Partitioning

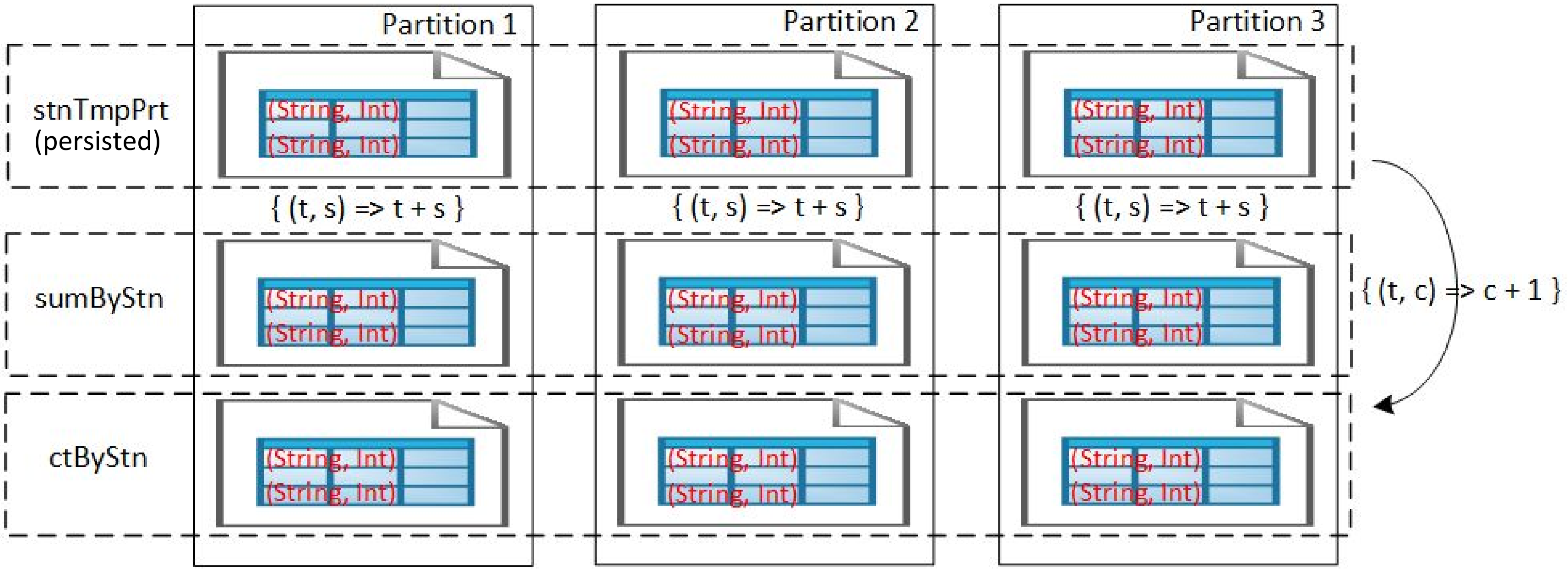
- Controls sets of keys residing on the same node
- Partition to minimize network traffic between nodes to improve performance
- Multiple partitioning options in Spark:
  - HashPartitioner
  - RangePartitioner
- Partitioning only useful only when RDD can be used in key-oriented operation
- Partitioning maintained for these operations:
  - all joins
  - all *ByKey() transformations
  - sort()
  - mapValues(), flatMapValues()
  - filter()

Hash Partitioning



stnTmpPrt = stnTmp.partitionBy(new HashPartitioner(3)).persist

reduceByKey() & foldByKey() with Partitioner

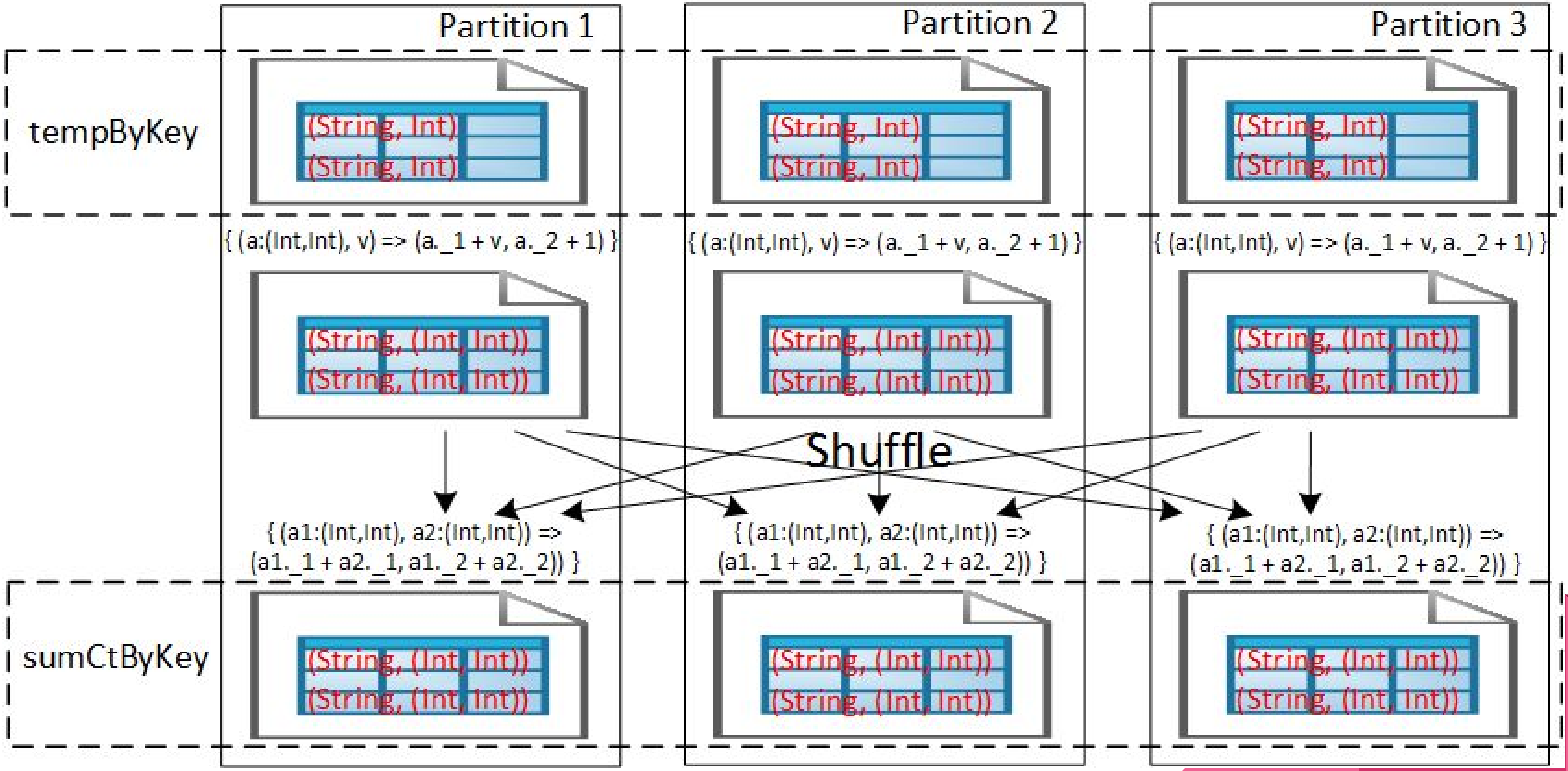


```
val sumByStn = stnTmpPrt.reduceByKey((t, s) => t + s)
val ctByStn = stnTmpPrt.foldByKey(0)((t, c) => c + 1)
```

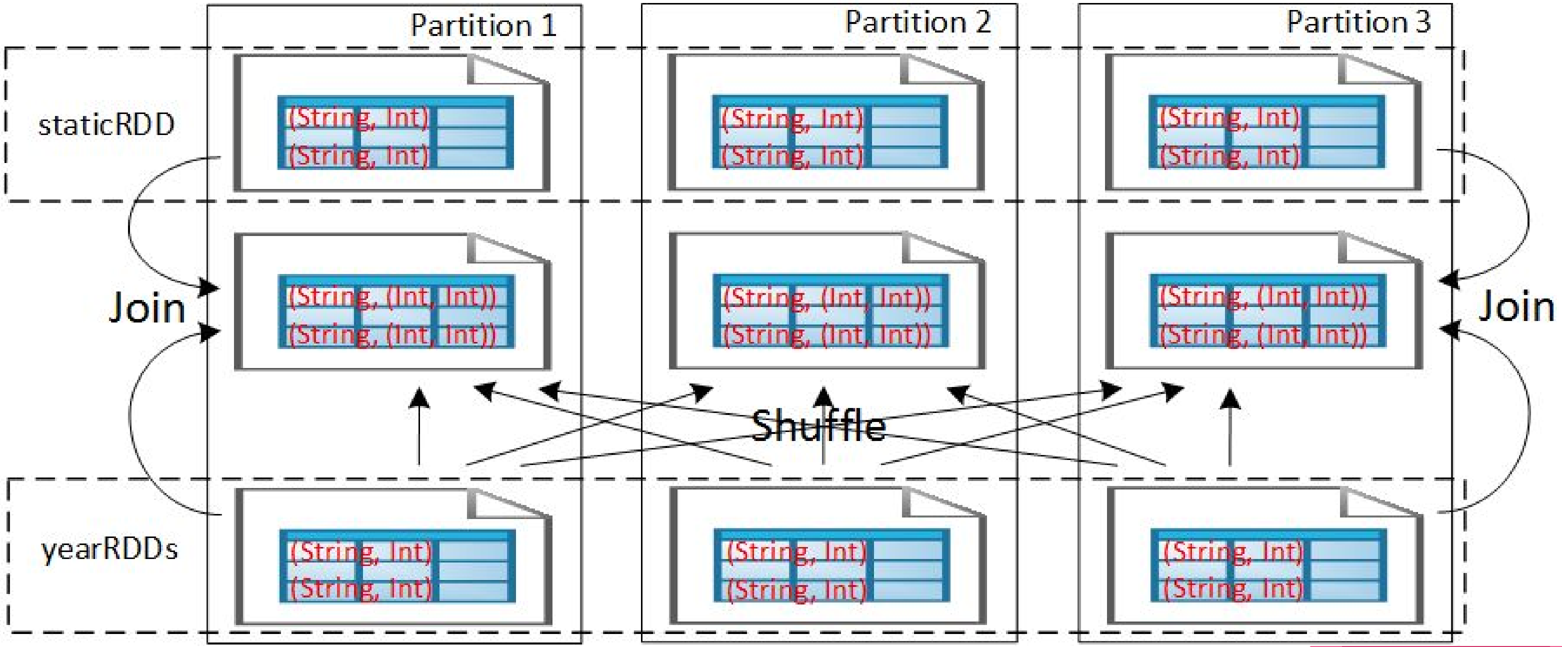
combineByKey()

```
val tempByKey = sc.textFile("input", 3)
    .map(line => line.split(","))
    .filter(fields => "TMAX".equals(fields(2)))
    .map(fields => (fields(0), Integer.parseInt(fields(3))))
val sumCtByKey = tempByKey.combineByKey((v:Int) => (v, 1),
    (a:(Int,Int), v) => (a._1 + v, a._2 + 1),
    (a1:(Int,Int), a2:(Int,Int)) => (a1._1 + a2._1, a1._2 + a2._2))
```

combineByKey()



Joins & Iterations with Partitioning



```
val staticRDD = yearRDDs(0).partitionBy(new HashPartitioner(100)).persist
for ( d <- 1 to 9)
    val runRDD = staticRDD.join(yearRdds(d)) // time series calc...
```

Resources

Installation

- http://spark.apache.org/docs/latest/
- http://spark.apache.org/docs/latest/hadoop-provided.html
- http://docs.scala-lang.org/tutorials/scala-with-maven.html#using-m2eclipse-scala35-for-eclipse-integration

General

- http://spark.apache.org/docs/latest/programming-guide.html
- Spark in Action. Manning
- Learning Spark. O'Reilly

Thank You !!

References

1. http://spark.apache.org/
2. Matei Zaharia. 2014. An Architecture for Fast and General Data Processing on Large Clusters. Association for Computing Machinery.
3. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. 2015. Learning Spark: Lightning-Fast Big Data Analytics. O'Reilly Media Inc.
4. Petar Zečević and Marko Bonaći. 2016. Spark in Action. Manning Publications.
5. http://spark.apache.org/docs/latest/programming-guide.html