Programming for Data Science at Scale

Distributed Key-Value Processing

THE UNIVERSITY of EDINBURGH

Amir Shaikhha, Fall 2024

Key-Value Pairs

- Single-node
  - Key-value pairs = Dictionaries
- Dictionaries are not the most commonly collections in single-node programs
- List/Arrays are most common

Distributed Key-Value Pairs

- Most common in big data processing
- Key design choice in MapReduce
  - Manipulating key-value pairs

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thou-

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

Distributed Key-Value Pairs

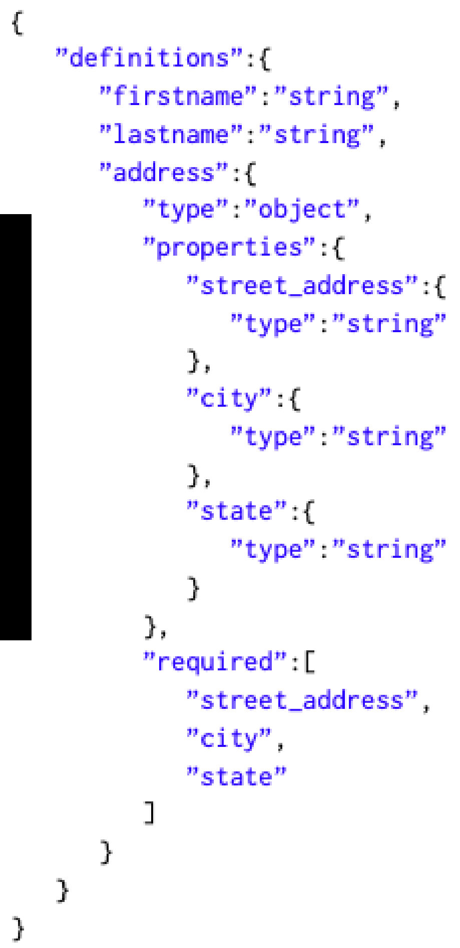
- Large datasets are often made up of complex nested data records
- To work with such datasets, it is often desirable to project down these nested data types into key-value pairs

JSON Example

- It may be disrable to create:

```
// city: String
val properties: RDD[(String, Property)]

case class Property(
       street: String,
       city: String,
       state: String)
```

- Where instances of this RDD are grouped by their cities



```
{
   "definitions":{
      "firstname":"string",
      "lastname":"string",
      "address":{
         "type":"object",
         "properties":{
            "street_address":{
               "type":"string"
            },
            "city":{
               "type":"string"
            },
            "state":{
               "type":"string"
            }
         },
         "required":[
            "street_address",
            "city",
            "state"
         ]
      }
   }
}
```

Pair RDDs

- Often when working with distributed data, it’s useful to organize data into key-value pairs.
- In Spark, distributed key-value pairs are called **Pair RDDs**.
- Pair RDDs allow you to
  - Act on each key in parallel
  - Regroup data across the network

Pair RDDs

- Such RDDs are treated specially by Spark
- Spark automatically adds a number of useful additional methods

```
def groupByKey(): RDD[(K, Iterable[V])] = ...
def reduceByKey(f: (V, V) => V): RDD[(K, V)] = ...
def join[W](other: RDD[(K, W)]): RDD[(K, (V,W))] = ...
```

Creating Pair RDDs

- Pair RDDs are most often created from existing non-pair RDDs

```
val rdd: RDD[Property] = ...

val pairRDD: RDD[(String, String)] =
  rdd.map(p => (p.city, p.street))
```

- Once created, you can use Pair-RDD-specific transformations

Transformations on Pair RDD

- `groupByKey`
- `reduceByKey`
- `mapValues`
- `keys`
- `join`
- `leftOuterJoin/rightOuterJoin`

Grouping in Scala Collections

- Recall `groupBy` of Scala collections

```
class List[T] {
  def groupBy[K](f: T => K): Map[K, Traversable[T]]
}
```

- Partitions this collection into a map of traversable collections according to some descriminator function

Scala Collections Example

- Let's group the below list of ages into "child", "adult", and "senior" categories.

```
val ages = List(2, 52, 44, 23, 17, 14, 12, 82, 51, 64)
val grouped = ages.groupBy({age =>
  if (age >= 18 && age < 65) "adult"
  else if (age < 18) "child"
  else "senior"
})
```

```
//grouped: scala.collection.immutable.Map[String,List[Int]] =
//Map(senior-> List(82), adult-> List(52, 44, 23, 51, 64),
//child-> List(2, 17, 14, 12))
```

Grouping in Pair RDDs

- Spark Pair-RDDs’ `groupByKey`
  - A `groupBy` on Pair RDDs specialized on grouping all values that have the same key
  - Thus, no argument is required

```
class PairRDD[K, V] {
  def groupByKey(): RDD[(K, Iterable[V])]
}
```

Spark Grouping Example

```
case class Event(organizer: String,
    name: String,
    budget: Int)

val eventsRdd = sc.parallelize(...)
  .map(event => (event.organizer, event.budget))

val groupedRdd = eventsRdd.groupByKey()
```

- If the key is organizer, what does this call do?
- Nothing ☺

Spark Grouping Example

```
case class Event(organizer: String,
    name: String,
    budget: Int)

val eventsRdd = sc.parallelize(...)
  .map(event => (event.organizer, event.budget))

val groupedRdd = eventsRdd.groupByKey()

groupedRdd.collect().foreach(println)
```

```
// (Prime Sound,CompactBuffer(42000))
// (Sportorg, CompactBuffer(23000, 12000, 1400))
// ...
```

Reduction in Pair RDDs

- Conceptually, reduceByKey can be thought of as a combination of groupByKey and reduce-ing on all the values per key.
- It's more efficient though, than using each separately.

```
class PairRDD[K, V] {
  def reduceByKey(f: (V, V) => V): RDD[(K, V)]
}
```

Pair RDD Reduction Example

```
case class Event(organizer: String,
    name: String,
    budget: Int)

val eventsRdd = sc.parallelize(...)
  .map(event => (event.organizer, event.budget))

val budgetsRdd = eventsRdd.reduceByKey(_ + _)

budgetsRdd.collect().foreach(println)
```

```
// (Prime Sound, 42000)
// (Sportorg, 36400)
// ...
```

Other Pair RDD operations

- `mapValues[U](f: V => U): RDD[(K, U)]`
  - Can be thought of a short-hand for:
    `rdd.map { case (x, y)=> (x, f(y))}`
  - Simply applies a function to only the values in a Pair RDD
- `countByKey(): Map[K, Long]`
  - Action
  - Count the number of elements per key in a Pair RDD

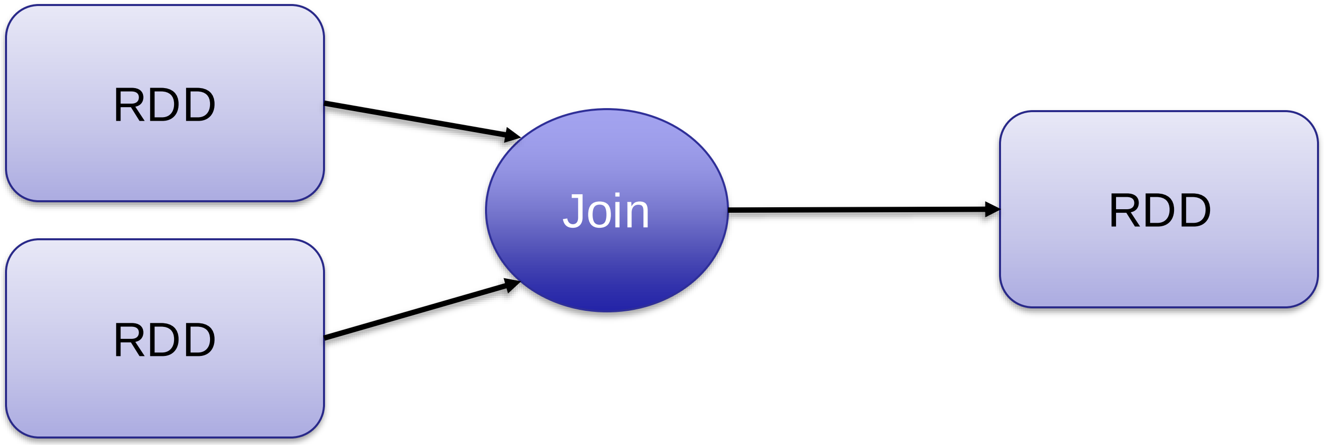
Pair RDD Averaging Example

```
// Calculate a K-V pair containing (budget, #events)
val intermediate = eventsRdd.mapValues (b => (b, 1) )
  .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))

val avgBudgets = intermediate.mapValues {
  case (budget, numberOfEvents) => budget / numberOfEvents
}

avgBudgets.collect().foreach(println)
```

```
// (Prime Sound, 42000)
// (Sportorg, 12133)
// ...
```

Joins on Pair RDDs

- Another transformation on Pair RDDs.
- They're used to combine multiple datasets.



```
class PairRDD[K, V] {
  def join[W](other: RDD[(K, W)]): RDD[(K, (V,W))]
}
```

Joins on Pair RDDs

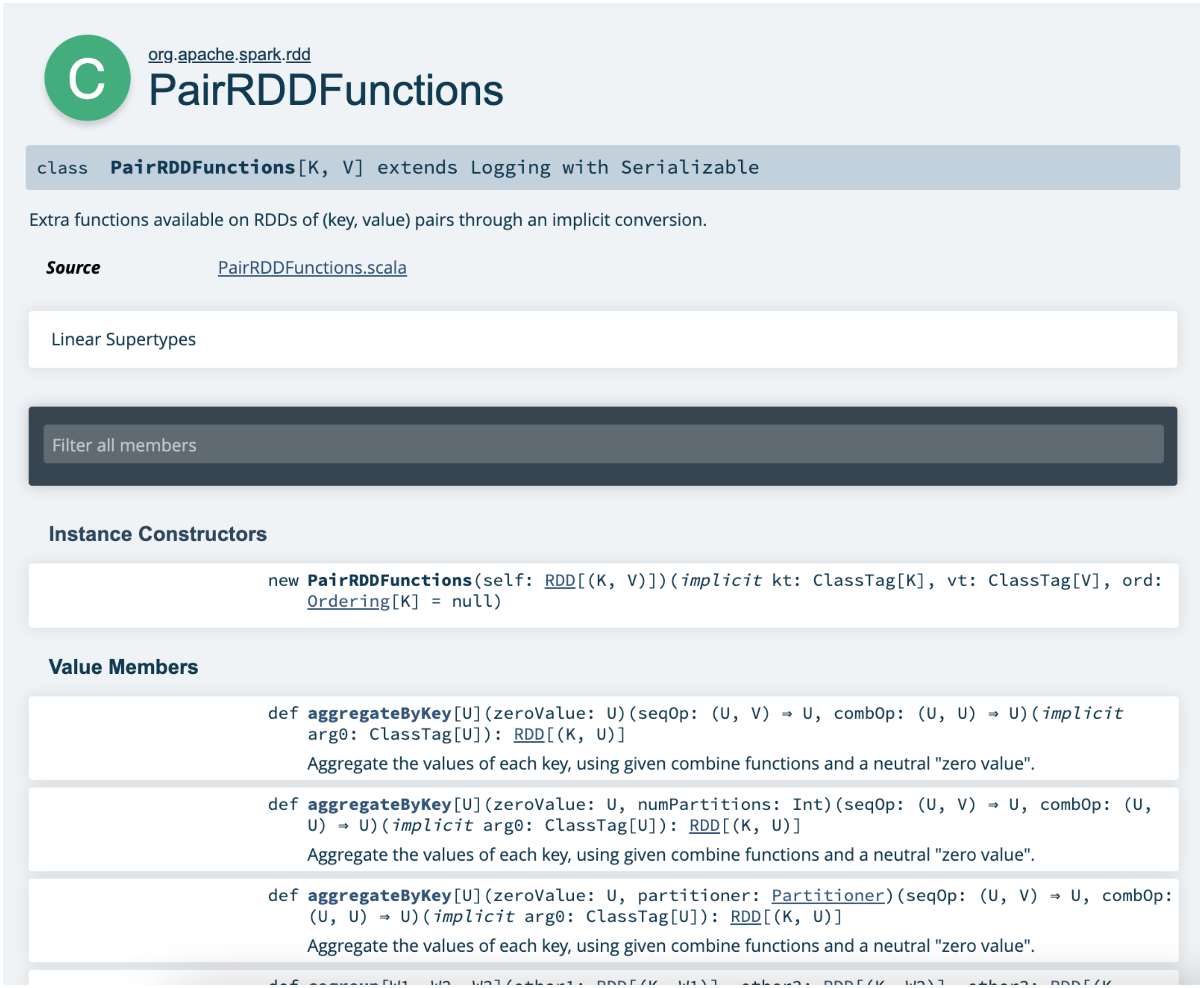
- They are one of the most commonly-used operations on Pair RDDs!
- What happens to the keys when two RDDs don’t contain the same key
- Two kinds
  - Inner joins (`join`)
  - Outer joins (`leftOuterJoin` / `rightOuterJoin`)

Pair RDD Join Example

```
case class Event(organizer:String,name:String,budget:Int)
case class Organizer(id: String, revenue: Int)

val eventsRdd = sc.parallelize(...)
  .map(event => (event.organizer, event.budget))
val orgRdd = sc.parallelize(...)
  .map(org => (org.id, org.revenue))

val joinRdd = eventsRdd.join(orgRdd)

joinRdd.collect().foreach(println)
```

```
// (Prime Sound, (42000, 2000000))
// (Sportorg, (23000, 4000000))
// (Sportorg, (12000, 4000000))
// (Sportorg, (1400, 4000000))
// ...
```

Other operations



org.apache.spark.rdd

PairRDDFunctions

```
class PairRDDFunctions[K, V] extends Logging with Serializable
```

Extra functions available on RDDs of (key, value) pairs through an implicit conversion.

Source PairRDDFunctions.scala

Linear Supertypes

Filter all members

Instance Constructors

```
new PairRDDFunctions(self: RDD[(K, V)])(implicit kt: ClassTag[K], vt: ClassTag[V], ord:
    Ordering[K] = null)
```

Value Members

```
def aggregateByKey[U](zeroValue: U)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit
    arg0: ClassTag[U]): RDD[(K, U)]
```

Aggregate the values of each key, using given combine functions and a neutral "zero value".

```
def aggregateByKey[U](zeroValue: U, numPartitions: Int)(seqOp: (U, V) ⇒ U, combOp: (U,
    U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]
```

Aggregate the values of each key, using given combine functions and a neutral "zero value".

```
def aggregateByKey[U](zeroValue: U, partitioner: Partitioner)(seqOp: (U, V) ⇒ U, combOp:
    (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]
```

Aggregate the values of each key, using given combine functions and a neutral "zero value".

https://spark.apache.org/docs/latest/api/scala/org/apache/spark/rdd/PairRDDFunctions.html

References

- Compulsory reading:
  - MapReduce [OSDI’04]:
    - MapReduce: Simplified data processing on large clusters
  - Spark [NSDI’12]
    - Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing
- Recommended reading:
  - YARN [SoCC’13]
    - The next generation of M/R
  - Dryad [EuroSys’07]
    - Generalized framework for data-parallel computations

QUESTIONS?