Formulaire - MAC (Big Data)

Leonard Cseres | January 21, 2026

Big Data: 3V + Véracité
- **Volume**: Too large for traditional DBs (> 1TB, distributed)
- **Vitesse**: Speed of data generation (batch vs stream)
- **Variété**: Structured, semi-structured, unstructured
- **Véracité**: Data quality (inconsistent, incomplete, ambiguous)

HDFS (Hadoop Distributed File System) Files split into blocks (64MB), distributed across cluster nodes. Blocks are replicated (3 copies by default) for fault tolerance.

MapReduce Pipeline
1. Read partitioned data (HDFS)
2. **Map**: extract/transform records → (key, value) pairs
3. **Shuffle & Sort**: group by key (automatic)
4. **Reduce**: aggregate values per key
5. Write results

Input/Output: bags of (key, value) pairs

MapReduce vs Spark MapReduce: Writes intermediate data to disk between Map and Reduce (for fault tolerance) → high latency
Spark: Keeps data immutable and in-memory. Instead of saving data, keeps DAG of transformations. Can replay DAG to recover from failures.
Spark is up to 100x faster in-memory, 10x faster on disk.

RDD (Resilient Distributed Dataset)
- **Resilient**: fault-tolerant via DAG replay
- **Distributed**: partitioned across cluster nodes
- **Dataset**: collection of typed elements

Properties: **immutable**, **in-memory**, **lazy evaluation**, **typed**, **partitioned**

Creating RDDs
```
// From collection
val rdd = sc.parallelize(List(1,2,3))

// From file
val rdd = sc.textFile("hdfs://...")
```

Spark Execution Model
- **Driver**: Main program, creates SparkContext, sends tasks
- **Executors**: Worker nodes, execute tasks, store data
- **Cluster Manager**: Allocates resources (YARN, Mesos)

Actions return results to the driver node.

Transformations (Lazy) Return new RDDs, not computed until action called.

Op	Description
`map(f)`	Transform each element
`filter(f)`	Keep elements where f is true
`flatMap(f)`	Map then flatten results
`distinct`	Remove duplicates
`sortBy(f)`	Sort by key function
`union(other)`	Union of two RDDs
`intersection`	Intersection of two RDDs
`groupBy(f)`	Group by function result

```
rdd.map(x => x * 2)
rdd.filter(x => x > 10)
rdd.flatMap(line => line.split(" "))
rdd.sortBy(_._2, ascending=false)
```

Actions (Eager) Trigger computation, return result to driver.

Op	Description
`collect()`	Return all elements to driver
`take(n)`	Return first n elements
`first()`	Return first element
`count()`	Count elements
`reduce(f)`	Aggregate all elements
`foreach(f)`	Apply f to each element
`saveAsTextFile`	Write to file system

Caching / Persistence RDDs are recomputed on each action by default. Use `persist()` or `cache()` to keep in memory for reuse.
```
val cached = rdd.filter(...).persist()
cached.count()  // computed & cached
cached.take(10) // uses cache (fast)
```

Pair RDDs (Key-Value) RDD of tuples `(K, V)`. Required for joins and groupBy operations.
```
// Create PairRDD from RDD
val kvRDD = rdd.map(x => (x.id, x.value))
```
▶ **CRITICAL**: `join` only works on PairRDDs!
▶ Always `map` to create (key, value) before join

PairRDD Transformations

Op	Description
`groupByKey()`	Group values by key → `(K, Iterable[V])`
`reduceByKey(f)`	Reduce values per key → `(K, V)`
`mapValues(f)`	Apply f only to values (keeps keys)
`keys`	Return RDD of keys only
`values`	Return RDD of values only
`countByKey()`	Action: count per key → `Map[K, Long]`
`join(other)`	Inner join → `(K, (V, W))`
`leftOuterJoin`	Left join → `(K, (V, Option[W]))`
`rightOuterJoin`	Right join → `(K, (Option[V], W))`

reduceByKey vs groupByKey
\+ `reduceByKey`: reduces locally **before** shuffle → less network traffic
− `groupByKey`: sends all data over network **then** groups

Always prefer `reduceByKey` over `groupByKey` + reduce!

Pattern: Word Count
```
rdd.flatMap(_.split(" "))
   .map(word => (word, 1))
   .reduceByKey(_ + _)
```

Pattern: Average by Key
```
rdd.map(x => (x.key, (x.value, 1.0)))
   .reduceByKey((a,b) =>
     (a._1 + b._1, a._2 + b._2))
   .mapValues(x => x._1 / x._2)
```

Pattern: Join Two RDDs Given `rdd1: RDD[A]` and `rdd2: RDD[B]` with common key field:
```
val kv1 = rdd1.map(x => (x.keyField, x))
val kv2 = rdd2.map(x => (x.keyField, x))
val joined = kv1.join(kv2)
// Result: (key, (A, B))
```

Pattern: Filter + Count + Sort
```
rdd.filter(_.type == "Book")
   .map(x => (x.store, 1))
   .reduceByKey(_ + _)
   .sortBy(_._2)  // sort BEFORE take!
   .take(100)
```

Pattern: Composite Key (Multi-field grouping)
```
// Average revenue per (region, year)
salesRDD.filter(_.amount > 100)
  .map(s => ((s.region, s.year),
             (s.revenue, 1.0)))
  .reduceByKey((a,b) =>
    (a._1 + b._1, a._2 + b._2))
  .map { case ((r,y), (sum,cnt)) =>
    (r, y, sum/cnt) }
```

Pattern: Inverted Index
```
// (docId, content) -> (word, List[docId])
docsRDD.flatMap { case (docId, content) =>
    content.split(" ").map(w => (w, docId))
  }.groupByKey()
```

Error 1: Join without PairRDD
```
// WRONG
studentsRDD.join(coursesRDD)
// CORRECT
studentsRDD.map(s => (s.courseId, s))
  .join(coursesRDD.map(c => (c.courseId, c)))
```
Error 2: take() before sortBy()
```
// WRONG - takes random 100, then sorts
.take(100).sortBy(_._2)
// CORRECT - sorts all, takes top 100
.sortBy(_._2).take(100)
```
Error 3: mapValues returning key
```
// WRONG - mapValues only transforms value
.mapValues(x => (x._1, x._2.size))
// CORRECT
.mapValues(x => x.size)
```

Scala Syntax
Variables:
```
val x = 5   // immutable (preferred)
var y = 5   // mutable (avoid)
```
Tuples:
```
val t = (1, "hello")  // Tuple2[Int,String]
t._1  // 1 (first element)
t._2  // "hello" (second element)
```
Case Classes (immutable, no `new`, comparable by value):
```
case class Person(id: Int, name: String)
val p = Person(1, "Alice")
p.id  // 1
```
Anonymous Functions:
```
x => x * 2          // explicit
_ * 2               // placeholder syntax
_ + _               // (a,b) => a + b
_._1                // x => x._1
```
Pattern Matching:
```
.map { case (k, v) => k + v }
.mapValues { case (sum, cnt) => sum/cnt }
```
Collection Methods:
```
list.sum      // sum of elements
list.size     // number of elements
list.mkString(", ")  // join to string
```

DataFrame API Higher-level API built on RDDs with schema and SQL-like operations.
```
// Convert RDD to DataFrame
val df = rdd.toDF("col1", "col2")
// Or with case class RDD
val df = caseClassRDD.toDF
```

DataFrame Operations
```
df.groupBy(col("region"), col("product"))
  .agg(
    min(col("price")) as "minPrice",
    max(col("quantity")) as "maxQty"
  )
  .where(col("maxQty") > col("minPrice"))
  .orderBy(col("maxQty").asc)
  .select(col("region"), col("product"),
          col("maxQty"))
  .show()
```
Key functions: `groupBy`, `agg`, `where/filter`, `orderBy`, `select`, `show`
Aggregations: `min`, `max`, `sum`, `avg`, `count`

Type 1: Compute aggregate per key (avg, sum, count)
- Use `mapValues(x => (x, 1))` + `reduceByKey` pattern

Type 2: Join two RDDs and aggregate
- Map both to PairRDD with common key
- Use `join`, then `map` to extract needed fields

Type 3: Filter, transform, sort, take top N
- Apply `filter` first
- Create PairRDD if grouping needed
- `sortBy` **before** `take`

Type 4: Debug/fix code
- Check: PairRDD before join?
- Check: sortBy before take?
- Check: mapValues signature correct?

Type 5: DataFrame query
- Use `groupBy` + `agg` for aggregations
- Use `where` for filtering after aggregation
- Use `orderBy` for sorting