

Spark and Resilient Distributed Datasets

Amir H. Payberah
Swedish Institute of Computer Science

`amir@sics.se`

April 24, 2014



- ▶ MapReduce greatly simplified **big data** analysis on large, unreliable clusters.
- ▶ But as soon as it got popular, users wanted more:
 - **Iterative** jobs, e.g., machine learning algorithms
 - **Interactive** analytics

- ▶ Both **iterative** and **interactive** queries need one thing that MapReduce lacks:

Motivation

- ▶ Both **iterative** and **interactive** queries need one thing that MapReduce lacks:

Efficient primitives for **data sharing**.

Motivation

- ▶ Both **iterative** and **interactive** queries need one thing that MapReduce lacks:

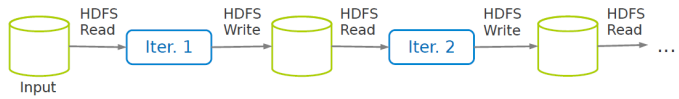
Efficient primitives for **data sharing**.

- ▶ In MapReduce, the only way to share data across jobs is stable **storage**, which is **slow**.
- ▶ **Replication** also makes the system slow, but it is necessary for **fault tolerance**.

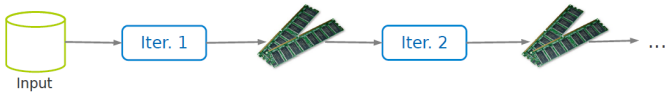
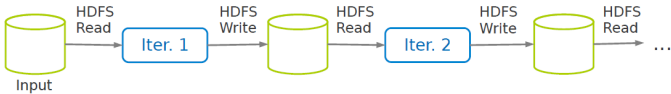
Proposed Solution

In-Memory Data Processing and Sharing.

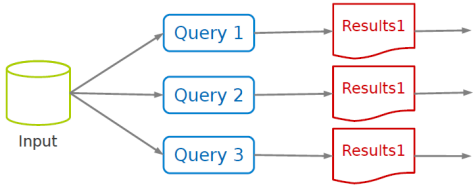
Example



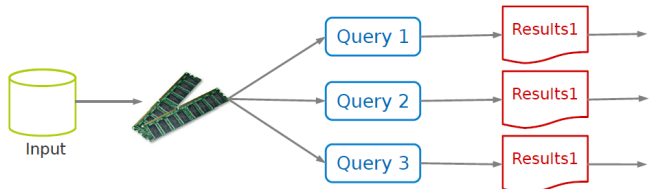
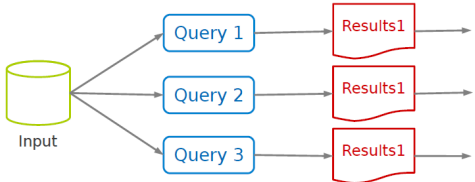
Example



Example



Example



Challenge

How to design a distributed memory abstraction that is both **fault tolerant** and **efficient**?

Challenge

How to design a distributed memory abstraction that is both **fault tolerant** and **efficient**?

Solution

Resilient Distributed Datasets (RDD)

Resilient Distributed Datasets (RDD) (1/2)

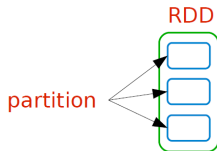
- ▶ A distributed memory abstraction.

Resilient Distributed Datasets (RDD) (1/2)

- ▶ A distributed memory abstraction.
- ▶ Immutable collections of objects spread across a cluster.

Resilient Distributed Datasets (RDD) (2/2)

- ▶ An **RDD** is divided into a number of **partitions**, which are **atomic** pieces of information.



- ▶ Partitions of an RDD can be stored on different **nodes** of a cluster.

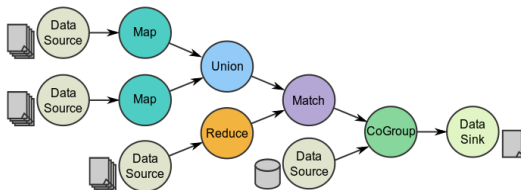
Programming Model

Spark Programming Model (1/2)

- ▶ Spark programming model is based on **parallelizable operators**.
- ▶ Parallelizable operators are **higher-order functions** that execute **user-defined functions** in parallel.

Spark Programming Model (2/2)

- ▶ A data flow is composed of any number of **data sources**, **operators**, and **data sinks** by connecting their inputs and outputs.
- ▶ **Job** description based on **directed acyclic graphs (DAG)**.



Higher-Order Functions (1/3)

- ▶ Higher-order functions: **RDDs** operators.
- ▶ There are two types of RDD operators: **transformations** and **actions**.

Higher-Order Functions (2/3)

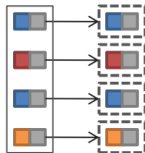
- ▶ **Transformations:** `lazy` operators that create `new` RDDs.
- ▶ **Actions:** launch a `computation` and return a `value` to the program or write data to the external storage.

Higher-Order Functions (3/3)

Transformations	<ul style="list-style-type: none"><i>map</i>($f : T \Rightarrow U$) : $RDD[T] \Rightarrow RDD[U]$<i>filter</i>($f : T \Rightarrow \text{Bool}$) : $RDD[T] \Rightarrow RDD[T]$<i>flatMap</i>($f : T \Rightarrow \text{Seq}[U]$) : $RDD[T] \Rightarrow RDD[U]$<i>sample</i>($\text{fraction} : \text{Float}$) : $RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling)<i>groupByKey</i>() : $RDD[(K, V)] \Rightarrow RDD[(K, \text{Seq}[V])]$<i>reduceByKey</i>($f : (V, V) \Rightarrow V$) : $RDD[(K, V)] \Rightarrow RDD[(K, V)]$<i>union</i>() : $(RDD[T], RDD[T]) \Rightarrow RDD[T]$<i>join</i>() : $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$<i>cogroup</i>() : $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (\text{Seq}[V], \text{Seq}[W]))]$<i>crossProduct</i>() : $(RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$<i>mapValues</i>($f : V \Rightarrow W$) : $RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning)<i>sort</i>($c : \text{Comparator}[K]$) : $RDD[(K, V)] \Rightarrow RDD[(K, V)]$<i>partitionBy</i>($p : \text{Partitioner}[K]$) : $RDD[(K, V)] \Rightarrow RDD[(K, V)]$
Actions	<ul style="list-style-type: none"><i>count</i>() : $RDD[T] \Rightarrow \text{Long}$<i>collect</i>() : $RDD[T] \Rightarrow \text{Seq}[T]$<i>reduce</i>($f : (T, T) \Rightarrow T$) : $RDD[T] \Rightarrow T$<i>lookup</i>($k : K$) : $RDD[(K, V)] \Rightarrow \text{Seq}[V]$ (On hash/range partitioned RDDs)<i>save</i>($\text{path} : \text{String}$) : Outputs RDD to a storage system, e.g., HDFS

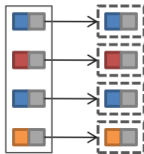
RDD Transformations - Map

- ▶ All pairs are **independently** processed.



RDD Transformations - Map

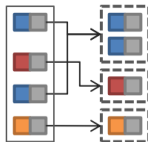
- ▶ All pairs are **independently** processed.



```
// passing each element through a function.  
val nums = sc.parallelize(Array(1, 2, 3))  
val squares = nums.map(x => x * x) // {1, 4, 9}  
  
// selecting those elements that func returns true.  
val even = squares.filter(x => x % 2 == 0) // {4}  
  
// mapping each element to zero or more others.  
nums.flatMap(x => Range(0, x, 1)) // {0, 0, 1, 0, 1, 2}
```

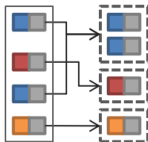
RDD Transformations - Reduce

- ▶ Pairs with **identical key** are grouped.
- ▶ Groups are independently processed.



RDD Transformations - Reduce

- ▶ Pairs with **identical key** are grouped.
- ▶ Groups are independently processed.



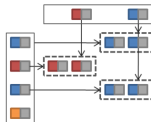
```
val pets = sc.parallelize(Seq(("cat", 1), ("dog", 1), ("cat", 2)))
```

```
pets.reduceByKey((x, y) => x + y)  
// {(cat, 3), (dog, 1)}
```

```
pets.groupByKey()  
// {(cat, (1, 2)), (dog, (1))}
```

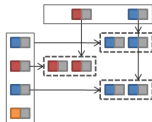
RDD Transformations - Join

- ▶ Performs an **equi-join** on the key.
- ▶ Join candidates are independently processed.



RDD Transformations - Join

- ▶ Performs an **equi-join** on the key.
- ▶ Join candidates are independently processed.



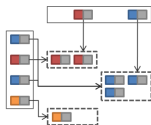
```
val visits = sc.parallelize(Seq(("index.html", "1.2.3.4"),
                              ("about.html", "3.4.5.6"),
                              ("index.html", "1.3.3.1")))

val pageNames = sc.parallelize(Seq(("index.html", "Home"),
                                   ("about.html", "About")))

visits.join(pageNames)
// ("index.html", ("1.2.3.4", "Home"))
// ("index.html", ("1.3.3.1", "Home"))
// ("about.html", ("3.4.5.6", "About"))
```

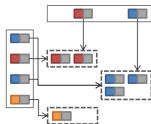
RDD Transformations - CoGroup

- ▶ Groups each **input on key**.
- ▶ Groups with identical keys are processed together.



RDD Transformations - CoGroup

- ▶ Groups each **input on key**.
- ▶ Groups with identical keys are processed together.



```
val visits = sc.parallelize(Seq(("index.html", "1.2.3.4"),
                               ("about.html", "3.4.5.6"),
                               ("index.html", "1.3.3.1")))

val pageNames = sc.parallelize(Seq(("index.html", "Home"),
                                   ("about.html", "About")))

visits.cogroup(pageNames)
// ("index.html", (("1.2.3.4", "1.3.3.1"), ("Home")))
// ("about.html", (("3.4.5.6"), ("About")))
```

RDD Transformations - Union and Sample

- ▶ **Union**: merges two RDDs and returns a single RDD using bag semantics, i.e., duplicates are not removed.
- ▶ **Sample**: similar to mapping, except that the RDD stores a random number generator seed for each partition to deterministically sample parent records.

Basic RDD Actions (1/2)

- ▶ Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))  
nums.collect() // Array(1, 2, 3)
```

Basic RDD Actions (1/2)

- ▶ Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))  
nums.collect() // Array(1, 2, 3)
```

- ▶ Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```


Basic RDD Actions (1/2)

- ▶ Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))  
nums.collect() // Array(1, 2, 3)
```

- ▶ Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

- ▶ Return the number of elements in the RDD.

```
nums.count() // 3
```

Basic RDD Actions (2/2)

- ▶ Aggregate the elements of the RDD using the given function.

```
nums.reduce((x, y) => x + y)  
or  
nums.reduce(_ + _) // 6
```

Basic RDD Actions (2/2)

- ▶ Aggregate the elements of the RDD using the given function.

```
nums.reduce((x, y) => x + y)  
or  
nums.reduce(_ + _) // 6
```

- ▶ Write the elements of the RDD as a text file.

```
nums.saveAsTextFile("hdfs://file.txt")
```

SparkContext

- ▶ **Main entry** point to Spark functionality.
- ▶ Available in **shell** as variable **sc**.
- ▶ In **standalone** programs, you should make your **own**.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._


val sc = new SparkContext(master, appName, [sparkHome], [jars])
```

SparkContext

- ▶ **Main entry** point to Spark functionality.
- ▶ Available in **shell** as variable **sc**.
- ▶ In **standalone** programs, you should make your **own**.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

val sc = new SparkContext(master, appName, [sparkHome], [jars])
```



local
local[k]
spark://host:port
mesos://host:port

- ▶ Turn a collection into an RDD.

```
val a = sc.parallelize(Array(1, 2, 3))
```

Creating RDDs

- ▶ Turn a collection into an RDD.

```
val a = sc.parallelize(Array(1, 2, 3))
```

- ▶ Load text file from local FS, HDFS, or S3.

```
val a = sc.textFile("file.txt")  
val b = sc.textFile("directory/*.txt")  
val c = sc.textFile("hdfs://namenode:9000/path/file")
```

Example (1/2)

- ▶ Count the lines containing SICS.

```
val file = sc.textFile("hdfs://...")
val sics = file.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_+_)
```


Example (1/2)

- ▶ Count the lines containing **SICS**.

```
val file = sc.textFile("hdfs://...")
val sics = file.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_+_)
```

Transformation

Transformation

Action

Example (2/2)

- ▶ Count the lines containing SICS.

```
val file = sc.textFile("hdfs://...")  
val count = file.filter(_.contains("SICS")).count()
```

Example (2/2)

- ▶ Count the lines containing SICS.

```
val file = sc.textFile("hdfs://...")  
val count = file.filter(_.contains("SICS")).count()
```



Transformation



Action

Example - Standalone Application (1/2)

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object WordCount {
  def main(args: Array[String]) {
    val sc = new SparkContext("local", "SICS", "127.0.0.1",
      List("target/scala-2.10/sics-count_2.10-1.0.jar"))
    val file = sc.textFile("...").cache()
    val count = file.filter(_.contains("SICS")).count()
  }
}
```

Example - Standalone Application (2/2)

► `sics.sbt`:

```
name := "SICS Count"

version := "1.0"

scalaVersion := "2.10.3"

libraryDependencies += "org.apache.spark" %% "spark-core" % "0.9.0-incubating"

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

Shared Variables (1/2)

- ▶ When Spark runs a function in parallel as a set of tasks on **different nodes**, it ships a **copy** of each **variable** used in the function to each task.
- ▶ Sometimes, a variable needs to be **shared across tasks**, or between **tasks and the driver** program.

Shared Variables (2/2)

- ▶ No updates to the variables are propagated back to the driver program.
- ▶ General read-write shared variables across tasks is inefficient.
 - For example, to give every node a copy of a large input dataset.
- ▶ Two types of shared variables: broadcast variables and accumulators.

Shared Variables: Broadcast Variables

- ▶ A **read-only** variable **cached** on each machine rather than shipping a copy of it with tasks.
- ▶ The broadcast values are not shipped to the nodes more than **once**.

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: spark.Broadcast[Array[Int]] = spark.Broadcast(b5c40191-...)

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```


Shared Variables: Accumulators

- ▶ They are only **added**.
- ▶ They can be used to implement **counters** or **sums**.
- ▶ Tasks running on the cluster can then add to it using the **+=** operator.

```
scala> val accum = sc.accumulator(0)
accum: spark.Accumulator[Int] = 0

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
...

scala> accum.value
res2: Int = 10
```

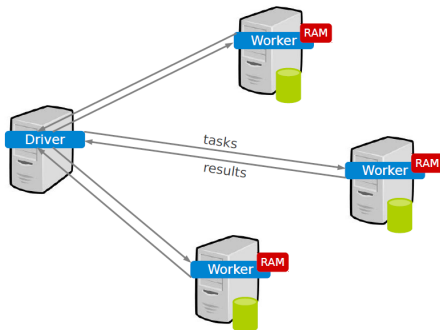
Execution Engine (SPARK)

- ▶ Spark provides a **programming interface** in **Scala**.
- ▶ Each RDD is represented as an **object** in Spark.



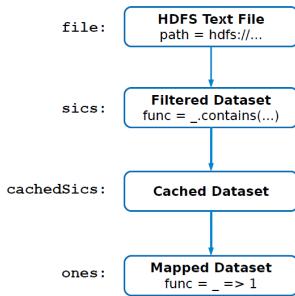
Spark Programming Interface

- ▶ A Spark application consists of a **driver program** that runs the user's **main** function and executes various **parallel operations** on a cluster.



Lineage

- ▶ **Lineage:** transformations used to build an RDD.
- ▶ **RDDs** are stored as a chain of objects capturing the **lineage** of each RDD.

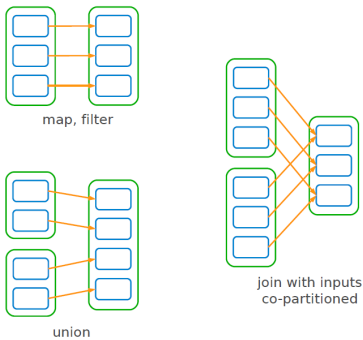


```
val file = sc.textFile("hdfs://...")
val sics = file.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_+_)
```

RDD Dependencies (1/3)

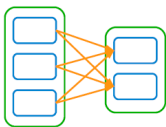
- ▶ Two types of dependencies between RDDs: **Narrow** and **Wide**.

RDD Dependencies: **Narrow** (2/3)

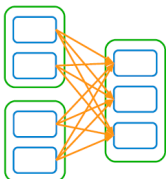


- ▶ **Narrow:** each partition of a parent RDD is used by **at most one** partition of the child RDD.
- ▶ Narrow dependencies allow **pipelined execution** on one cluster node: a map followed by a filter.

RDD Dependencies: **Wide** (3/3)



groupByKey

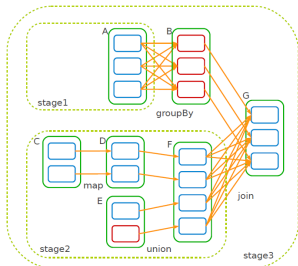


Join with inputs not
co-partitioned

- ▶ **Wide**: each partition of a parent RDD is used by **multiple** partitions of the child RDDs.

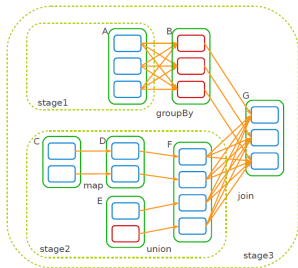
Job Scheduling (1/2)

- ▶ When a user runs an **action** on an RDD: the scheduler builds a **DAG** of **stages** from the RDD **lineage** graph.
- ▶ A **stage** contains as many **pipelined transformations** with **narrow dependencies**.
- ▶ The **boundary** of a stage:
 - **Shuffles** for wide dependencies.
 - Already **computed partitions**.



Job Scheduling (2/2)

- ▶ The scheduler launches **tasks** to compute **missing partitions** from each **stage** until it computes the target RDD.
- ▶ Tasks are assigned to machines based on data **locality**.
 - If a task needs a **partition**, which is available in the **memory** of a node, the task is sent to that node.



RDD Fault Tolerance (1/3)

- ▶ RDDs maintain **lineage** information that can be used to **reconstruct** lost partitions.
- ▶ **Logging lineage** rather than the **actual data**.
- ▶ **No replication**.
- ▶ Recompute only the **lost partitions** of an RDD.

Job Scheduling (2/3)

- ▶ The intermediate records of **wide dependencies** are **materialized** on the nodes holding the **parent** partitions: to **simplify** fault recovery.
- ▶ If a task fails, it will be re-ran on another node, as long as its **stages parents are available**.
- ▶ If some **stages** become **unavailable**, the tasks are submitted to compute the **missing partitions in parallel**.

RDD Fault Tolerance (3/3)

- ▶ Recovery may be **time-consuming** for RDDs with **long lineage chains** and **wide dependencies**.
- ▶ It can be helpful to **checkpoint** some RDDs to stable storage.
- ▶ Decision about which data to checkpoint is **left to users**.

Memory Management (1/2)

- ▶ If there is **not enough space in memory** for a new computed RDD partition: a partition from the **least recently used** RDD is evicted.
- ▶ Spark provides three options for storage of persistent RDDs:
 - ① In **memory** storage as **deserialized** Java objects.
 - ② In **memory** storage as **serialized** Java objects.
 - ③ On **disk** storage.

Memory Management (2/2)

- ▶ When an RDD is **persisted**, each node stores any **partitions** of the RDD that it computes in memory.
- ▶ This allows future actions to be much **faster**.
- ▶ Persisting an RDD using **persist()** or **cache()** methods.
- ▶ Different **storage levels**:
 - `MEMORY_ONLY`
 - `MEMORY_AND_DISK`
 - `MEMORY_ONLY_SER`
 - `MEMORY_AND_DISK_SER`
 - `MEMORY_ONLY_2`, `MEMORY_AND_DISK_2`, etc.

- ▶ Applications **suitable** for RDDs
 - **Batch applications** that apply the **same operation** to **all elements** of a dataset.

- ▶ Applications **not suitable** for RDDs
 - Applications that make **asynchronous fine-grained updates** to shared state, e.g., storage system for a web application.

- ▶ RDD: a distributed memory abstraction that is both **fault tolerant** and **efficient**
- ▶ Two types of operations: **Transformations** and **Actions**.
- ▶ RDD fault tolerance: **Lineage**

Questions?