Spark and Spark SQL

Amir H. Payberah
amir@sics.se

SICS Swedish ICT

What is Big Data?

Big Data

> ... everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it.
>
> \- Dan Ariely

Big Data

> Big data is the data characterized by 4 key attributes: volume, variety, velocity and value.
>
> \- Oracle

Big Data

> Big data is the data characterized by 4 key attributes: volume, variety, velocity and value.
>
> \- Oracle

Buzzwords

DevOps Borat

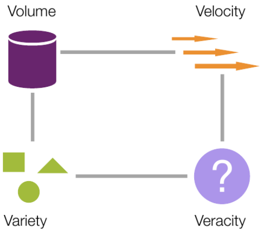
@DEVOPS_BORAT

Small Data is when is fit in RAM. Big Data is when is crash because is not fit in RAM.

2/6/13, 8:22 AM

The Four Dimensions of Big Data

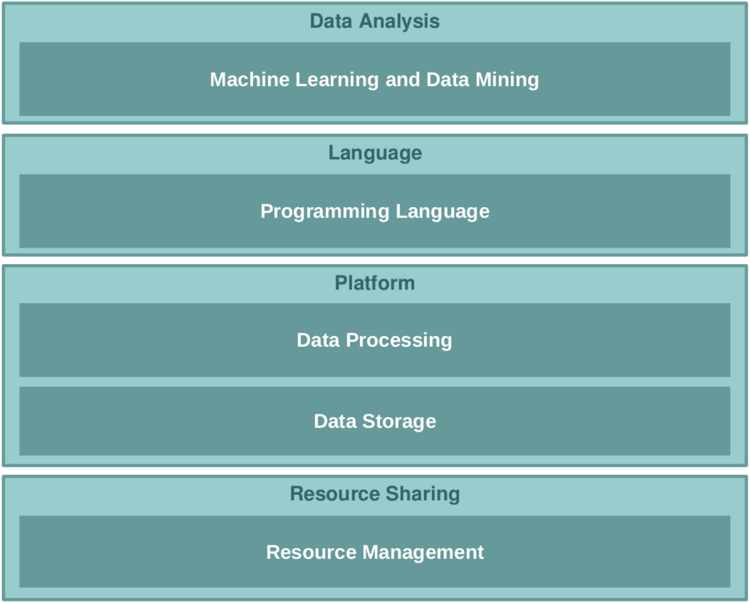
- Volume: data size
- Velocity: data generation rate
- Variety: data heterogeneity
- This 4th V is for Vacillation: Veracity/Variability/Value



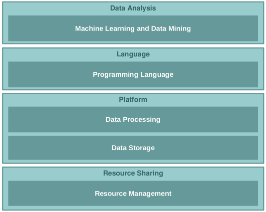
How To Store and Process Big Data?

Scale Up vs. Scale Out

The Big Data Stack



Data Analysis



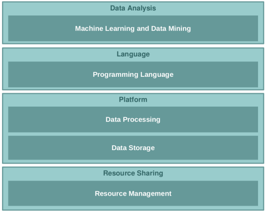
Programming Languages

Programming Languages

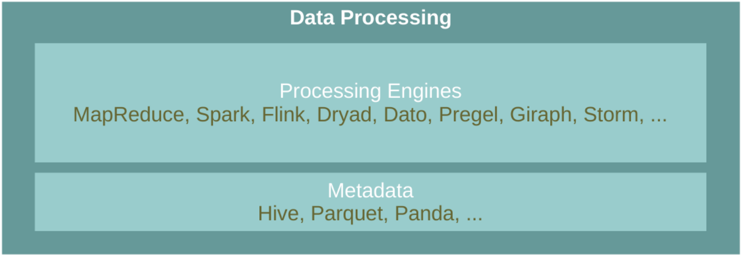
Imperative Languages
Scala, Python, Java, R, StreamIt, ...

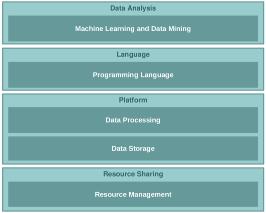
Declarative Languages
Hive, Pig, Spark SQL, CQL, HiPal, ...

Visual Languages
SQuAl, ...

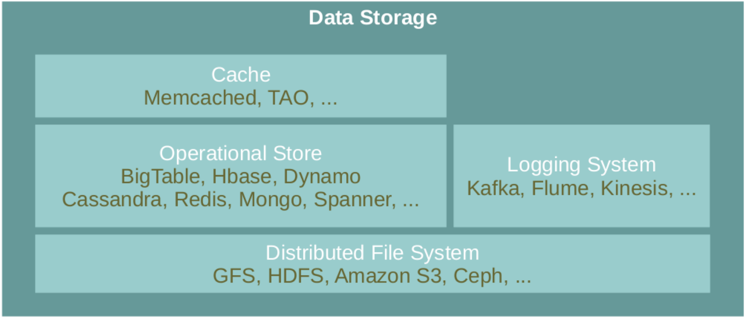


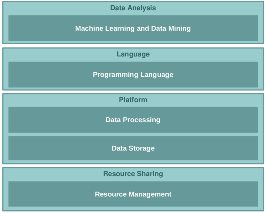
Platform - Data Processing



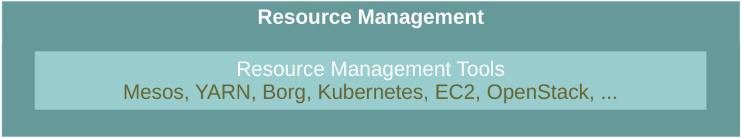


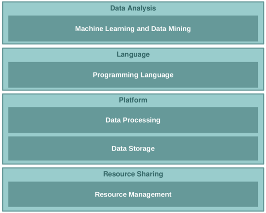
Platform - Data Storage



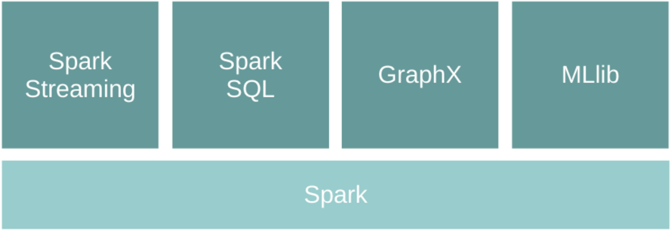


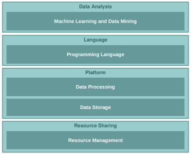
Resource Management





Spark Processing Engine

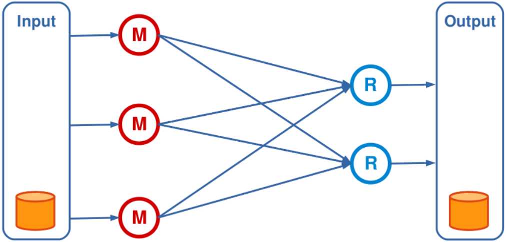




Why Spark?

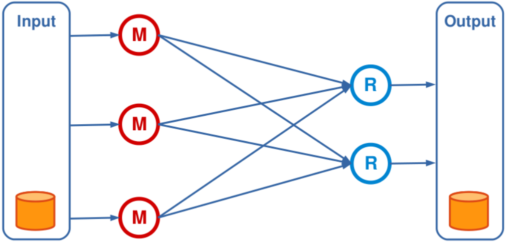
Motivation (1/4)

- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage.



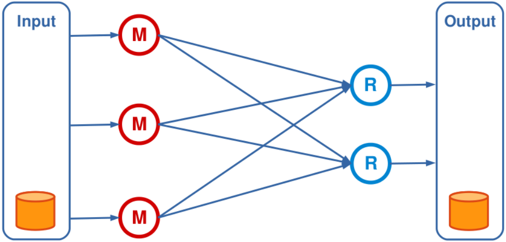
Motivation (1/4)

- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage.
- Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures.



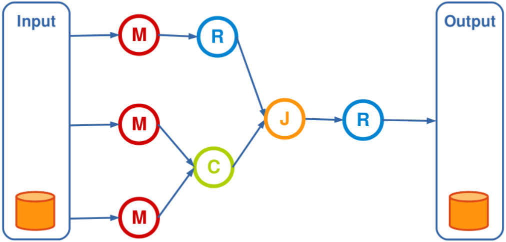
Motivation (1/4)

- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage.
- Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures.
- E.g., MapReduce



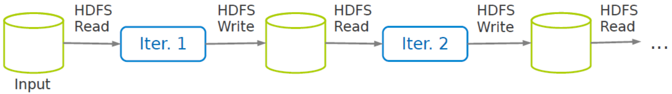
Motivation (2/4)

- MapReduce programming model has not been designed for complex operations, e.g., data mining.



Motivation (3/4)

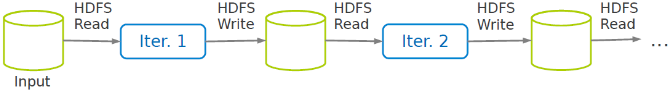
- Very expensive (slow), i.e., always goes to disk and HDFS.



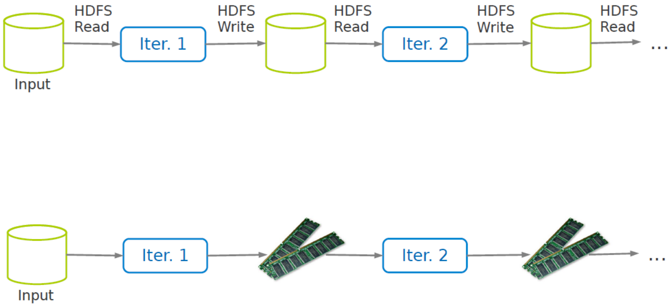
Motivation (4/4)

- Extends MapReduce with more operators.
- Support for advanced data flow graphs.
- In-memory and out-of-core processing.

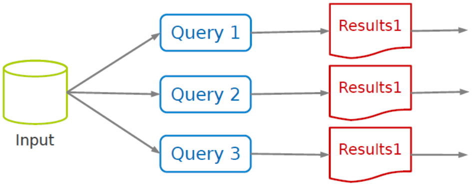
Spark vs. MapReduce (1/2)



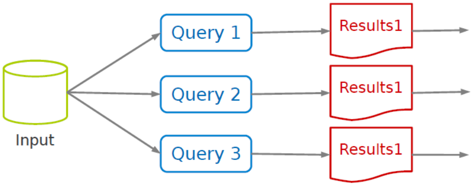
Spark vs. MapReduce (1/2)

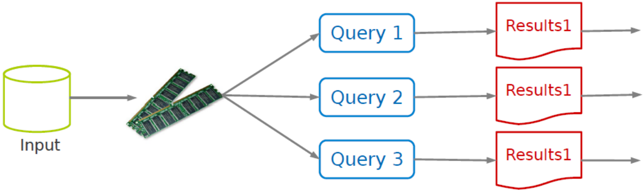


Spark vs. MapReduce (2/2)



Spark vs. MapReduce (2/2)





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.
- Immutable collections of objects spread across a cluster.
  - Like a `LinkedList <MyObjects>`

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.

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.
- Built through coarse grained transformations, e.g., `map`, `filter`, `join`.

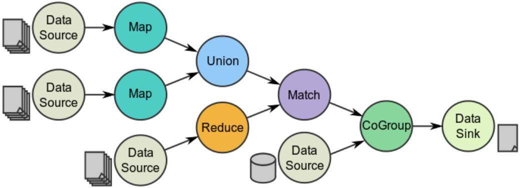
Resilient Distributed Datasets (RDD) (2/2)

- 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.
- Built through coarse grained transformations, e.g., `map`, `filter`, `join`.
- Fault tolerance via automatic rebuild (no replication).

Programming Model

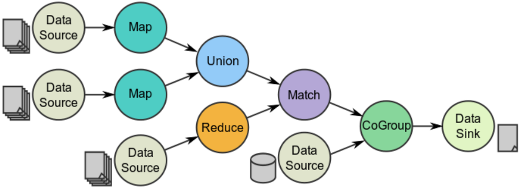
Spark Programming Model

- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.



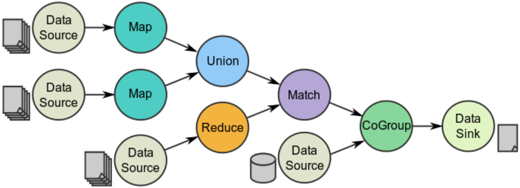
Spark Programming Model

- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.
- Operators are higher-order functions that execute user-defined functions in parallel.



Spark Programming Model

- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.
- Operators are higher-order functions that execute user-defined functions in parallel.
- Two types of RDD operators: transformations and actions.



RDD Operators (1/2)

- Transformations: lazy operators that create new RDDs.

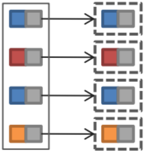
- Actions: lunch a computation and return a value to the program or write data to the external storage.

RDD Operators (2/2)

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

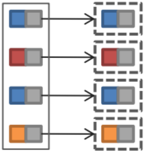
RDD Transformations - Map

- All pairs are independently processed.



RDD Transformations - Map

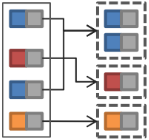
- All pairs are independently processed.



```scala
// 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(_ % 2 == 0) // {4}
```

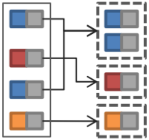
RDD Transformations - Reduce

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



RDD Transformations - Reduce

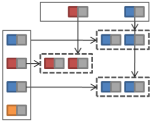
- Pairs with identical key are grouped.
- Groups are independently processed.



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

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

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

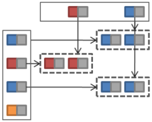
RDD Transformations - Join

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



RDD Transformations - Join

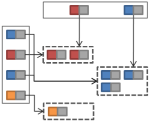
- Performs an equi-join on the key.
- Join candidates are independently processed.



```
val visits = sc.parallelize(Seq(("h", "1.2.3.4"),
                                ("a", "3.4.5.6"),
                                ("h", "1.3.3.1")))

val pageNames = sc.parallelize(Seq(("h", "Home"),
                                   ("a", "About")))

visits.join(pageNames)
  // ("h", ("1.2.3.4", "Home"))
  // ("h", ("1.3.3.1", "Home"))
  // ("a", ("3.4.5.6", "About"))
```

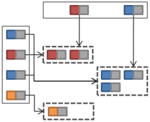
RDD Transformations - CoGroup

- 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(("h", "1.2.3.4"),
                                ("a", "3.4.5.6"),
                                ("h", "1.3.3.1")))

val pageNames = sc.parallelize(Seq(("h", "Home"),
                                   ("a", "About")))

visits.cogroup(pageNames)
  // ("h", (("1.2.3.4", "1.3.3.1"), ("Home")))
  // ("a", (("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.
- Only one SparkContext may be active per JVM.

```
// master: the master URL to connect to, e.g.,
// "local", "local[4]", "spark://master:7077"
val conf = new SparkConf().setAppName(appName).setMaster(master)

new SparkContext(conf)
```

Creating RDDs

- Turn a collection into an RDD.

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

Creating RDDs

- Turn a collection into an RDD.

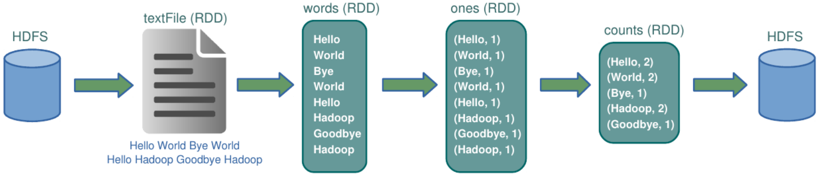
```
val a = sc.parallelize(Array(1, 2, 3))
```

- 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

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

val words = textFile.flatMap(line => line.split(" "))
val ones = words.map(word => (word, 1))
val counts = ones.reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```



Example 2

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

Example 2

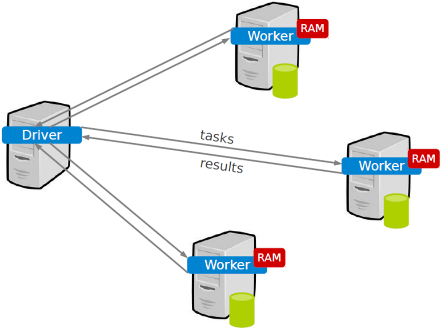
```
val textFile = sc.textFile("hdfs://...")
val sics = textFile.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_ + _)
```

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

Execution Engine

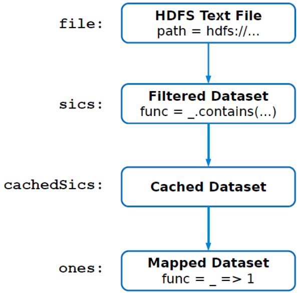
Spark Programming Interface

- 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.

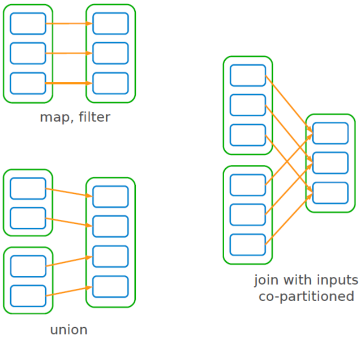



```scala
val file = sc.textFile("hdfs://...")
val sics = file.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_+_)
```

RDD Dependencies (1/3)

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

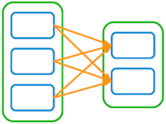
RDD Dependencies: Narrow (2/3)

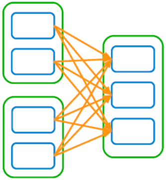


- Narrow: each partition of a parent RDD is used by at most one partition of the child RDD.

- Narrow dependencies allow pipelined execution on one cluster node, e.g., a `map` followed by a `filter`.

RDD Dependencies: Wide (3/3)

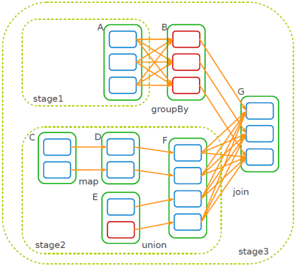




- Wide: each partition of a parent RDD is used by multiple partitions of the child RDDs.

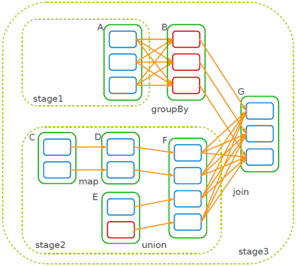
Job Scheduling (1/2)

- 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.



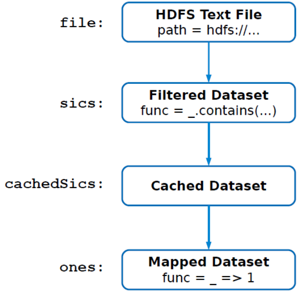
Job Scheduling (2/2)

- 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.



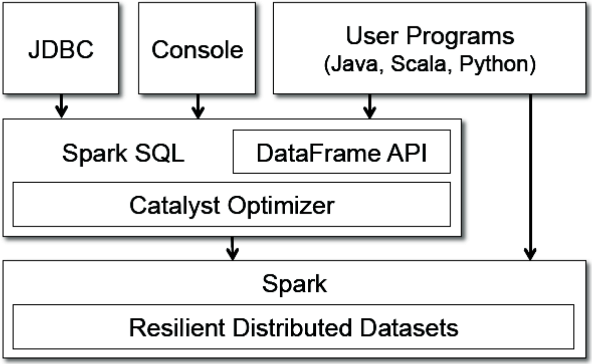
RDD Fault Tolerance

- Logging lineage rather than the actual data.
- No replication.
- Recompute only the lost partitions of an RDD.



Spark SQL

Spark and Spark SQL

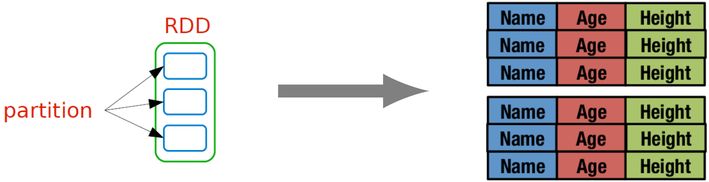


DataFrame

- A DataFrame is a distributed collection of rows
- Homogeneous schema.
- Equivalent to a table in a relational database.

Adding Schema to RDDs

- Spark + RDD: functional transformations on partitioned collections of opaque objects.
- SQL + DataFrame: declarative transformations on partitioned collections of tuples.



Creating DataFrames

- The entry point into all functionality in Spark SQL is the `SQLContext`.

- With a `SQLContext`, applications can create DataFrames from an existing RDD, from a Hive table, or from data sources.

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val df = sqlContext.read.json(...)
```

DataFrame Operations (1/2)

- Domain-specific language for structured data manipulation.

```
// Show the content of the DataFrame
df.show()
// age  name
// null Michael
// 30   Andy
// 19   Justin

// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// name
// Michael
// Andy
// Justin
```

DataFrame Operations (2/2)

- Domain-specific language for structured data manipulation.

```
// Select everybody, but increment the age by 1
df.select(df("name"), df("age") + 1).show()
// name    (age + 1)
// Michael null
// Andy    31
// Justin  20

// Select people older than 21
df.filter(df("age") > 21).show()
// age name
// 30  Andy

// Count people by age
df.groupBy("age").count().show()
// age  count
// null 1
// 19   1
// 30   1
```

Running SQL Queries Programmatically

- Running SQL queries programmatically and returns the result as a DataFrame.
- Using the `sql` function on a `SQLContext`.

```
val sqlContext = ...  // An existing SQLContext
val df = sqlContext.sql("SELECT * FROM table")
```

Converting RDDs into DataFrames

- Inferring the schema using reflection.

```
// Define the schema using a case class.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile(...).map(_.split(","))
               .map(p => Person(p(0), p(1).trim.toInt)).toDF()
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext
    .sql("SELECT name, age FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are DataFrames.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
teenagers.map(t => "Name: " + t.getAs[String]("name")).collect()
         .foreach(println)
```

Data Sources

- Supports on a variety of data sources.
- A DataFrame can be operated on as normal RDDs or as a temporary table.
- Registering a DataFrame as a table allows you to run SQL queries over its data.

and more ...

Advanced Programming

Shared Variables

- 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.
- General read-write shared variables across tasks is inefficient.
- Two types of shared variables: accumulators and broadcast variables.

Accumulators (1/2)

- Aggregating values from worker nodes back to the driver program.
  - Example: counting events that occur during job execution.
- Worker code can add to the accumulator with its += method.
- The driver program can access the value by calling the value property on the accumulator.

```
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
```

Accumulators (2/2)

- How many lines of the input file were blank?

```
val sc = new SparkContext(...)

val file = sc.textFile("file.txt")
val blankLines = sc.accumulator(0)

// Create an Accumulator[Int] initialized to 0
val callSigns = file.flatMap(line => {
  if (line == "") {
    blankLines += 1 // Add to the accumulator
  }

  line.split(" ")
})
```

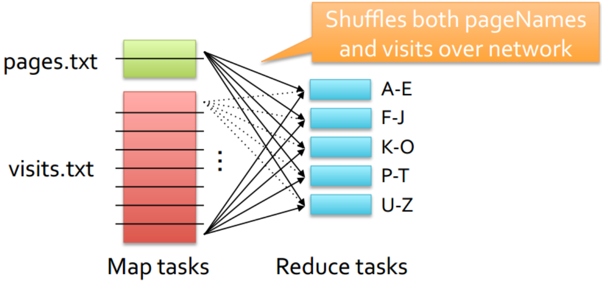
Broadcast Variables (1/4)

- The broadcast values are sent to each node only once, and should be treated as read-only variables.
- The process of using broadcast variables can access its value with the `value` property.

```
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)
```

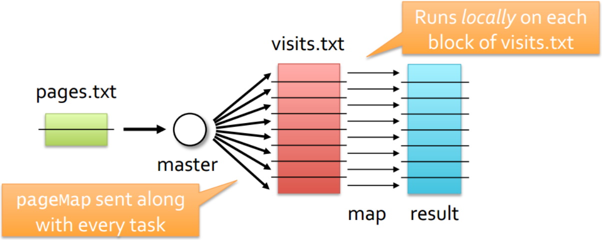
Broadcast Variables (2/4)

```
// Load RDD of (URL, name) pairs
val pageNames = sc.textFile("pages.txt").map(...)

// Load RDD of (URL, visit) pairs
val visits = sc.textFile("visits.txt").map(...)

val joined = visits.join(pageNames)
```



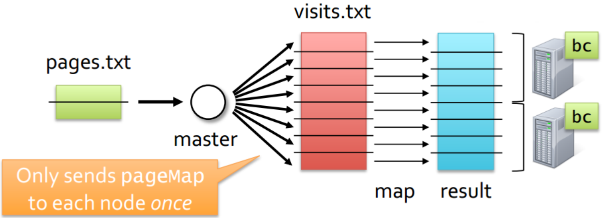
Broadcast Variables (3/4)

```
// Load RDD of (URL, name) pairs
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap()

// Load RDD of (URL, visit) pairs
val visits = sc.textFile("visits.txt").map(...)

val joined = visits.map(v => (v._1, (pageMap(v._1), v._2)))
```



Broadcast Variables (4/4)

```
// Load RDD of (URL, name) pairs
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap()
val bc = sc.broadcast(pageMap)

// Load RDD of (URL, visit) pairs
val visits = sc.textFile("visits.txt").map(...)

val joined = visits.map(v => (v._1, (bc.value(v._1), v._2)))
```



Summary

Summary

- Dataflow programming
- Spark: RDD
- Two types of operations: Transformations and Actions.
- Spark execution engine
- Spark SQL: DataFrame

Questions?