732A54 / TDDE31
Big Data Analytics

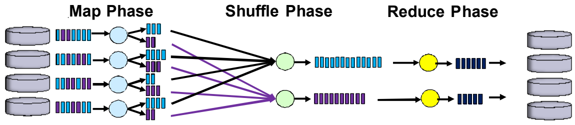
Introduction to Spark

Christoph Kessler
IDA, Linköping University

Christoph Kessler, IDA,
Linköpings universitet.

Recall: MapReduce Programming Model

- Designed to operate on LARGE distributed input data sets stored e.g. in HDFS nodes
- Abstracts from parallelism, data distribution, load balancing, data transfer, fault tolerance
- Implemented in **Hadoop** and other frameworks
- Provides a high-level parallel programming construct (= a skeleton) called **MapReduce**
  - A generalization of the data-parallel *MapReduce* skeleton of Lect. 1
  - Covers the following algorithmic design pattern:

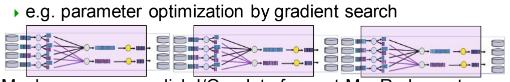


From MapReduce to Spark

MapReduce
- is for large-scale computations matching the *MapReduce* pattern,
- with input, intermediate and output data stored in secondary storage

By chaining multiple MapReduce steps, we can emulate *any* distributed computation.

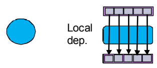
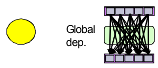
Limitations
- For complex computations composed of *multiple* MapReduce steps
  - E.g. iterative computations
    - e.g. parameter optimization by gradient search



  - → Much unnecessary disk I/O – data for next MapReduce step could remain in main memory or even cache memory
  - → Data blocks used multiple times are read multiple times from disk
  - → Bad data locality across subsequent Mapreduce phases
- Sharing of data only in secondary storage
  - Latency can be too long for interactive analytics
- Fault tolerance by replication of data – more I/O to store copies → slow

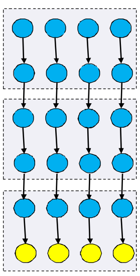
Splitting the MapReduce Construct into Simpler Operations – 2 Main Categories:

- **Transformations**: Elementwise operations, fully parallelizable
  - Mostly variants of **Map** and reading from distributed file
- **Actions**: Operations with internally global dependence structure
  - Mostly variants of **Reduce** and writing back to non-distr. file / to master

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$ $filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$ $flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$ $sample(fraction : \mathrm{Float})$ $groupByKey()$ $reduceByKey(f : (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$ $union()$ $join()$ $cogroup()$ $crossProduct()$ $mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$ $sort(c : \mathrm{Comparator[K]})$ $partitionBy(p : \mathrm{Partitioner[K]})$	
Actions	$count()$ $collect()$ $reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$ $lookup(k : \mathrm{K})$ $save(path : \mathrm{String})$	

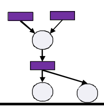
RDD transformations and actions available in Spark. Seq[T] denotes a sequence of elements of type T.
Table source: Zaharia *et al.*: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.

Spark Idea: Data Flow Computing in Memory

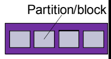


Instead of calling subsequent rigid MapReduce steps, the Spark programmer describes the overall **data flow graph** of how to compute all intermediate and final results from the initial input data

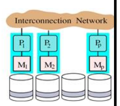
- And then "pushes the button" for computing (= *materializing* the results) according to the data flow graph → *Lazy evaluation*
- More like declarative, functional programming
- Gives more flexibility to the scheduler
  - Better data locality
  - Keep data in memory as capacity permits, can skip unnecessary disk storage of temporary data
- No replication of data blocks for fault tolerance - in case of task failure (worker failure), **recompute** it from available, earlier computed data blocks according to the data flow graph
  - Needs a container data structure for operand data that "knows" how its data blocks are to be computed: the RDD



Resilient Distributed Datasets (RDDs)

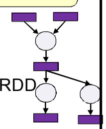


- **Containers for operand data** passed between parallel operations
  - *Read-only* (after construction) collection of data objects
  - Partitioned and distributed across workers (cluster nodes)
  - Materialized on demand from construction description
  - Can be rebuilt if a partition (data block) is lost
  - By default, cached **in main memory** – not persistent (in secondary storage) until written back



- **Construction of new RDDs:**
  - By reading in from a file e.g. in HDFS
  - By partitioning and distributing a non-distributed collection (e.g., array) previously residing on one node ("*scatter*")
  - By a *Map* operation: *A* → List(*B*) (elementwise transformation, filtering, …) applied on another RDD
  - **Changing persistence state of a RDD:**
    - By a **caching** hint for data to be reused – if enough space in memory
    - By **materializing** (persisting, saving) to a file (and discarding its copy in memory)

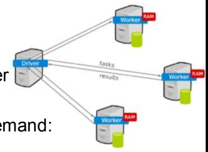
data = [1, 2, 3, 4, 5]
distData = sc.**parallelize**(data)



cachedData = distdata.**cache**()

distdata.**saveAsTextFile**(...)

Parallel Operations on RDDs



Spark execution model:

- **Driver program** (sequential) runs on host / master
- Operations on RDDs run on **workers**
- Collect data from workers to driver program on demand:

Parallel Collect Operations on RDDs:

- **Reduce**
  - Combine RDD elements using an associative binary function to produce a (scalar) result at the driver program
  - Key-value pairs to reduce over are grouped by key, as in MapReduce
  - Initially no shuffle&sort phase → no grouped reduction as in MapReduce
- **Collect**
  - Send all elements of the RDD to the driver program (”*gather*”)
- **Foreach**
  - Pass each RDD element through a user-provided function
  - Not producing another RDD (difference from Map/Filter)
  - Might be used e.g. for copying data to another system

distData = sc.**parallelize**(data)
distData.**collect**()

Classification of RDD Operations

- **Transformations**: Lazy, parallelizable
  - Mostly variants of Map and reading from file
- **Actions**: Materialization points (”push the button”)
  - Mostly variants of Reduce and writing back to file/master

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

RDD transformations and actions available in Spark. Seq[T] denotes a sequence of elements of type T.
Table source: Zaharia *et al.*: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.

Shared Variables

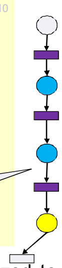
- **Broadcast Variables**
  - Replicated shared variables – 1 copy on each worker
  - Read-only for workers
  - For global data needed by all workers, e.g. filtering parameters, lookup table
- **Accumulator Variables**
  - Residing on driver program process
  - Workers can not read, only add their contributions using an associative operation
  - Good for implementing counters and for global sum

Example: Text Search

- Count lines containing errors in a large log file stored in HDFS

Python code adapted from Zaharia *et al.* 2010

```
// Create a RDD from file:
file = sc.textFile("hdfs://...")

// Filter operation to create RDD containing lines with "ERROR":
errs = file.filter( lambda line: line.find("ERROR")>=0 )

// Map each line to a 1:
ones = errs.map( lambda word: (word, 1) )

// Add up the 1's using Reduce:
count = ones.reduce( lambda x, y: x+y )
```



The ”**lineage**” of RDDs leading to the result *count*

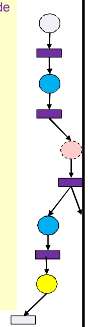
- RDDs *errs* and *ones* are lazy RDDs that are never materialized to secondary storage.
- Call to **reduce** triggers computation of *ones*, which triggers computation of *errs*, which triggers reading blocks from the file.

Example: Text Search, with reuse of *errs*

- Count lines containing errors in a large log file stored in HDFS

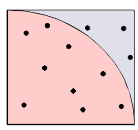
Python pseudocode

```
// Create a RDD from file:
file = sc.textFile("hdfs://...")

// Filter operation to create RDD containing lines with "ERROR":
errs = file.filter( lambda line: line.find("ERROR")>=0 )

// Cache hint that errs will be reused in another operation:
cachedErrs = errs.cache();

// Map each line to a 1:
ones = cachedErrs.map( lambda word: (word, 1) )

// Add up the 1's using Reduce:
count = ones.reduce( lambda x, y: x+y )
```



Example: Pi Calculation

- Stochastic approximation of Pi:
  - A random point (x,y) in [0,1]x[0,1] is located within quarter unit cycle iff $x^2 + y^2 < 1$

1

1

```
def sample(p):
    x, y = random(), random()
    return 1 if x*x + y*y < 1 else 0

count = sc.parallelize( xrange(0, NUM_SAMPLES) ) \
              .map(sample) \
              .reduce( lambda a, b: a + b)

print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Create a RDD containing all indexes 0, …, NUM_SAMPLES-1

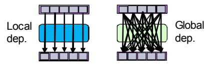
Example: Logistic Regression

- Iterative classification algorithm to find a hyperplane that best separates 2 sets of data points
- Gradient descent method:
  - Start at a random normal-vector (hyperplane) *w*
  - In each iteration, add to *w* an error-correction term (based on the *gradient*) that is a function of *w* and the data points, to improve *w*

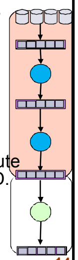
Scala pseudocode, adapted from Zaharia et al., 2010

```
// Read points from a text file and cache them:
points = sc.textFile(...).map(parsePoint).cache()
// Initialize w to random D-dimensional vector:
w = Vector.random(D)
// Run multiple iterations to update w:
for (i <- 1 to NUMBER_OF_ITERATIONS) {
    grad = sc.accumulator( new Vector(D) )
    for (p <- points) { // Runs in parallel:
        val s = (1/(1+exp(-p.y*(w dot p.x))-1) * p.y
        grad += s * p.x  // remotely add contribution to gradient value
    }
    w -= grad.value     // correction of w
}
```


Spark Execution Model



- Depending on the kind of operations, the data dependences between RDDs in the lineage graph can be **local** (elementwise) or **global** (shuffle-like)
- When a user runs an *action* on an RDD, the Spark scheduler builds a DAG of *stages* from the RDD lineage graph.
- A **stage** contains a contiguous subDAG of as many as possible operations with *local* (element-wise) dependencies between RDDs
  - The boundary of a stage is thus defined by
    - Operations with global dependencies
    - Already computed (materialized) RDD partitions.
- Execution of the operations within a stage is **pipelined**
  - intermediate results forwarded in memory
- The scheduler launches **tasks** to workers (cluster nodes) to compute missing partitions from each stage until it computes the target RDD.
- Tasks are assigned to nodes based on data locality.
  - If a task needs a partition that is available in the memory of a node, the task is sent to that node.



Spark Performance

Results from original paper on Spark 2010:

- Spark can outperform Hadoop by 10x in iterative machine learning jobs
- Interactive query of a 39GB data set in < 1s

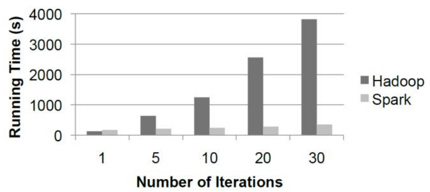


Figure 2: Logistic regression performance in Hadoop and Spark.

Image source: M. Zaharia *et al.*, 2010. © ACM

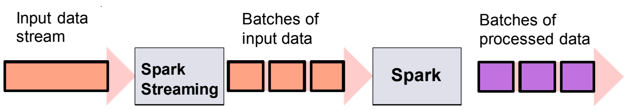
Using Spark

- Spark can run atop HDFS, but other implementations also exist
- Language bindings exist for Scala, Java, Python (PySpark)
  - Some minor restrictions for Python
- Spark Context object
  - The main entry point to Spark functionality
  - Represents connection to a Spark cluster
  - PySpark context *sc* is up and running from start
  - Create your own Spark context object for stand-alone applications
    - sc = **new** pyspark.SparkContext(*master*, *applName*, [sparkHome], [...])

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

Spark Streaming

- Extension of the core Spark API for scalable, high-throughput, fault-tolerant stream processing of live data streams.
- **Discretized stream** or **DStream**
  - High-level abstraction representing a continuous stream of data.
  - Internally: A continuous series of RDDs



Transformations on DStreams

- **map**(*func*), **flatMap**(*func*), **filter**(*func*) – return a new DStream with **map** etc. applied to all its elements
- **repartition**(), **union**(other_stream)
- **count**() – returns a new DStream of single-element RDDs containing the number of elements in each RDD of the source DStream
- **reduce**(func), **reduceByKey**() – aggregate each RDD of the source Dstream and return a new Dstream of single-element RDDs
- **join** (other_stream) – joins 2 streams of (K,V) and (K,W) pairs to a stream of (K,(V,W)) pairs
- **transform**(func) – apply arbitrary RDD-to-RDD function to each RDD in the source DStream
- …

Spark Streaming Example

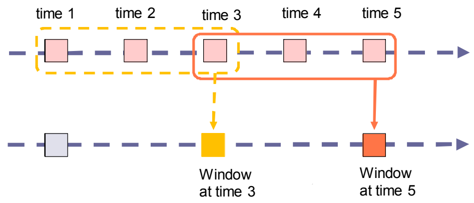
```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working threads and batch interval of 1 second:
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)

# Create a DStream that will connect to TCP hostname:port, like localhost:9999, as source:
lines = ssc.socketTextStream("localhost", 9999)

# Split each line into words:
words = lines.flatMap( lambda line: line.split(" ") )
# Count each word in each batch:
pairs = words.map( lambda word: (word, 1) )
wordCounts = pairs.reduceByKey( lambda x, y: x + y )
# Print the first ten elements of each RDD generated in this DStream to the console:
wordCounts.pprint()

ssc.start()              # Start the computation
ssc.awaitTermination()   # Wait for the computation to terminate
```

Run on local host, alt. cluster name

DStream of lines

Spark Streaming: Windowing

- Can define a sliding window over a source DStream



Window length (here 3)
Slide length (here 2)
→ Overlap size (here 1)

Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.

```
# Example: Reduce last 30 seconds of data, every 10 seconds:
windowedWordCounts = \
  pairs.reduceByKeyAndWindow( lambda x, y: x + y, lambda x, y: x - y, 30, 10 )
```


References

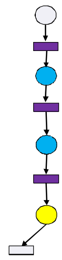
- M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, I. Stoica: Spark: Cluster Computing with Working Sets. Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (*HotCloud'10*), 2010, ACM.
  - See also: M. Zaharia *et al*.: Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56-65, Nov. 2016.
- Apache Spark: http://spark.apache.org
- A. Nandi: *Spark for Python Developers*. Packt Publishing, 2015.

Questions for Reflection

- Why can MapReduce emulate any distributed computation?
- For a Spark program consisting of 2 subsequent Map computations, show how Spark execution differs from Hadoop/MapReduce execution.
- Given is a file containing just integer numbers. Write a Spark program that adds them up.
- Write a wordcount program for Spark.
  - Solution proposal (from spark.apache.org):

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap( lambda line: line.split(" ") ) \
             .map( lambda word: (word, 1) ) \
             .reduceByKey( lambda a, b: a + b )
counts.saveAsTextFile("hdfs://...")
```

  - Note – there exist many variants for formulating this.



- Modify the wordcount program by only considering words with at least 4 characters.